UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Alenka Starc

CENITEV PROJEKTOV RAZVOJA PROGRAMSKE OPREME

Magistrsko delo

Mentor: prof. dr. Franc Solina

Ljubljana 2004

Zahvala

Zahvaljujem se mentorju prof. dr. Francu Solini, za njegovo pomoč in nasvete pri izdelavi naloge.

Zahvaljujem se podjetju Hermes Softlab, ki mi je omogočilo delati na tako zanimivih projektih in je imelo tudi razumevanje za moj študij. Hvala Gregorju Smrekarju, ki je predlagal obdelano področje, kot eno izmed možnih tem in Urošu Sajku za pomoč pri zbiranju podatkov o preteklih projektih.

Hvala družini in prijateljem za razumevanje in vzpodbudo.

Povzetek

V nalogi je predstavljen kratek pregled tematike ocenjevanja potrebnega dela na projektih razvoja programske opreme. Glavni cilj naloge je prikazati način ocenjevanja dela, ki se uporablja v primeru razvoja velikih programskih projektov. Avtorica te naloge sodeluje pri razvoju produkta Data Protector kot projektni vodja in kot vodja dela ekipe, ki dela na razvoju produkta. Drugi pomemben cilj naloge je raziskati, kako si lahko pri ocenjevanju dela pomagamo z metodo parametričnega ocenjevanja. V okviru tega cilja je predstavljen parametričen model COCOMO II, prav tako pa je ta model prilagojen za projekte razvoja produkta Data Protector. Na osnovi prilagojenega modela je narejena tudi parametrična ocena trenutnega projekta. V prilogi je predstavljena tudi predlagana struktura arhiva za projekte Data Protector, ki se lahko uporabi kot osnova v okviru ocenjevanja s pomočjo parametričnega modela.

Abstract

The reader of the thesis is first introduced with the short overview of the software estimation process. The main objective of the thesis is to describe the necessary processes involved in the estimation process of big software projects. The author of the thesis works as project and team manager on the Data Protector product. The second important objective of the thesis is to explore the parametric estimation method. COCOMO II parametric model is presented and calibrated to the local environment of Data Protector projects. Based on the calibrated model the parametric estimation is done for the currently active project. Proposed structure of the Data Protector archive, which can be used as the base of the future parametric estimation, is described in the appendix.

KAZALO VSEBINE

KAZALO TABEL

KAZALO SLIK

1. Uvod

Programska oprema postaja v svetu vse bolj pomembna na vseh področjih življenja. Računalniki in programska oprema nas spremljajo tako rekoč povsod, tudi takrat, ko jih ne vidimo. Programska oprema upravlja čedalje bolj kompleksne sisteme. Od nje pričakujemo, da bo učinkovita, enostavna za uporabo, integrirana v ostale sisteme... Vse te zahteve povečujejo kompleksnost programske opreme, projekti postajajo vse večji, obenem pa pričakujemo, da bodo končani v predvidenem času in v okviru planiranega proračuna. Glavna naloga projektnega vodje programskega projekta je, da projekt uspešno pripelje na cilj, torej da izpolni začetne zahteve projekta, pri tem poskrbi, da je kvaliteta končnega izdelka ustrezna in da je projekt končan v predvidenem roku in ob predvidenih stroških. Seveda še tako uspešen projektni vodja, ki vodi izkušeno ekipo, ne more izpolniti pričakovanj, če so pričakovanja nerealna.

Kako postaviti prave cilje projektu, takšne, ki jih bomo lahko z dobrim delom in predanostjo projektu dosegli? Gotovo je to eno izmed težjih vprašanj industrije programske opreme. Vprašanje ni enostavno, saj moramo pri tem napovedati prihodnost, ne znamo pa pogledati v stekleno kroglo in iz nje potegniti odgovora. Kako torej odgovoriti na vprašanje, koliko časa in koliko denarja potrebujemo da končamo projekt? Če bo naš odgovor preveč optimističen, bomo imeli težave, ker

ciljev ne bomo dosegli. V primeru, da za izvedbo projekta zahtevamo preveliko časa in denarja, obstaja velika verjetnost, da naš projekt sploh ne bo dobil zelene luči za začetek. Odgovorov, pravzaprav ocen, je veliko in seveda so zelo različni. Gotovo pa so odvisni od tega, komu bomo zastavili vprašanje.

Ponavadi je ocena programerjev vedno preveč optimistična. Brooks [14] ugotavlja, da je možen vzrok ta, da so programerji večinoma mladi ljudje, ki gledajo na svet bolj optimistično. To je lahko eden izmed možnih vzrokov. Postavi pa se vprašanje, kako programer oceno sploh razume. Ali mu pomeni končano delo dokončan produkt ali končano kodiranje njegovega modula. Dokončano kodiranje modula pa seveda ne pomeni, da smo z delom pri koncu. Modul bo potrebno tudi obsežno testirati, zanj bo potrebno napisati dokumentacijo. Ponavadi je potrebno modul tudi integrirati med ostale module. Vse to pa zahteva dodaten čas in dodatno delo, na katerega programer v trenutku, ko smo mu postavili vprašanje, ni pomislil. Takrat je seveda mrzlično izbiral algoritme in premišljeval, katero tehniko bo uporabil za to, da bo njegov modul narejen.

Kaj pa, če postavi oceno projektni vodja? Kako bo on pristopil k delu? Gradil bo na preteklih izkušnjah in skušal poiskati projekt, ki je trenutnemu najbolj podoben. Pa obstajata dva popolnoma podobna projekta? Tudi če na prvi pogled projekt izgleda na las podoben prejšnjemu, je paralelo zelo težko potegniti. Gotovo se v čem projekt tudi razlikuje. Postavi se vprašanje ali bo projektni vodja znal te razlike identificirati, oceniti njihovo pomembnost in primerno prilagoditi oceno.

Kaj pa, če postavi oceno specialist za postavljanje ocen, katerega glavna naloga je, da skrbi za postavljanje ocen znotraj podjetja. Je njegovo delo podobno delu kontrolorja, ki meri kako hitro delavci za tekočim trakom opravijo posamezne korake, ki v končni fazi pripeljejo do končnega produkta? Pravzaprav res, vendar pa zopet naletimo na problem. V tovarni lahko naredijo na dan nekaj deset, mogoče tudi nekaj tisoč kosov produkta in tako je vzorec, na katerem kontrolor postavlja svoje ocene, na katerih temelji njegovo planiranje, primerno velik. Koliko časa pa traja, da končamo programski projekt? Podatkov o programskih projektih je na voljo bistveno manj, saj projekti ponavadi trajajo vsaj nekaj mesecev, velik projekti pa trajajo tudi nekaj let. Druga težava je tudi v tem, da projekti niso enaki, potrebno jih je po njihovih lastnostih grupirati v skupine, določiti kateri parametri vplivajo na izvedbo projekta... Razlika med najhitrejšim in najpočasnejšim delavcem za tekočim trakom lahko predstavlja nekaj odstotkov. Razlika med najbolj hitrim in najbolj počasnim

programerjem pa je bistveno večja. Brooks [14] navaja podatek iz raziskave Sackmana, Eriksona in Granta, da je razmerje produktivnosti v skupini izkušenih programerjev 1:10. Tako je potrebno precej časa, da analitik dobro razume lastnosti projekta, okolje v katerem projekt poteka, pa tudi čas in metode, ki se uporabljajo za izvedbo projekta.

Metode, ki nas pripeljejo do ocen so zelo različne. Vsem je skupno nekaj. Vse gradijo na znanju iz preteklosti. Tako ocena programerja, ki je vlekel paralele med svojimi prejšnjimi programerskimi podvigi, kot ocena projektnega vodje, ki je ravno tako poskušal s primerjavo novega projekta s prejšnjimi. Težava je le v tem, da ljudje ponavadi pozabimo na probleme in si od preteklih izkušenj rajši zapomnimo lepše plati. Tako je programer gotovo pozabil, da je presedel na prejšnjem projektu kar nekaj noči in koncev tedna ob iskanju napak. Pozabil je tudi upoštevati, da je zadnji mesec na projektu večere prebil ob pisanju tehnične in pregledovanju uporabniške dokumentacije.

Tudi projektni vodja je poskušal poiskati analogijo s predhodnimi projekti. Kljub temu, da ponavadi gledajo na svet projektni vodje z bolj realističnimi očmi, pa je seveda vprašanje, če tudi on ni nekaj pomembnih podatkov zanemaril. Seveda je pod pritiskom svojih nadrejenih, ki pričakujejo od njega ocene, ki bodo napovedale prihod novega projekta v čim krajšem času in ne prevelikih stroških. Drugače mogoče projekta sploh ne bo. Projektni vodja pa si projekt želi. Hoče, da se projekt začne, da se spet znajde v vrtincu projektnega vodenja in ne da gleda ob strani in čaka, kdaj bodo sprejeli njegove pretirane ocene. Torej bo tudi on poskušal dati bolj pozitivno oceno o trajanju projekta, čeprav bo prav on tisti, ki bo v končni fazi moral trpeti posledice, tako da bo opravičeval zaostanek projekta pri svojih nadrejenih in pa seveda tudi stranki.

Specialist za ocenjevanje trajanja projekta se bo zadeve najbrž lotil nekoliko drugače. Skušal bo uporabiti podatke, ki so zbrani in opisujejo pretekle projekta. S pomočjo teh podatkov bo poskusil postaviti model, ki bo čim bolj zadovoljivo opisoval projekte, ki se odvijajo v njegovem podjetju. Za oceno trenutnega projekta pa bo moral narediti obširne analize dejavnikov, ki vplivajo na izvedbo projekta. Kljub temu, da je pristop, ki ga bo uporabil bolj formalen, pa še ne zagotavlja popolnih rezultatov.

Kdaj pa lahko zastavimo vprašanja o ocenah projekta in kdaj dobimo zadovoljivo natančen odgovor? Ponavadi potrebujemo oceno na samem začetku. Takrat se

odločamo, ali je projekt prava odločitev ali ne. Na samem začetku ocene lahko znatno odstopajo od realnosti, saj imamo bistveno premalo podatkov. Prva ocena, ki jo lahko podamo, je v trenutku, ko so nam na voljo prve specifikacije projekta. Torej moramo za oceno poznati osnovne zahteve. Bolj ko so specifikacije popolne in izčrpne, lažje bomo na njihovi osnovi podali oceno. Problemi, ki se kasneje lahko pojavijo so: nepravilna interpretacija specifikacije, različno razumevanje specifikacije stranke, kasnejše spremembe zahtev, itd. Projekt je živ, poln dinamičnega dogajanja, tako da je potrebno tudi ocene projekta prilagajati ves čas, ko projekt teče. Na ta način dosežemo, da je projekt pod kontrolo, da vemo, kje smo in ali smo dosegli prave cilje ali ne. Ocene se lahko v vsaki fazi projekta popravijo glede na trenutno situacijo, saj vsaka faza prinese nove informacije, ki bolj natančno opredeljujejo zahteve projekta. Nič ni narobe, če se ocene spremenijo. To je le dokaz, da nismo postavili oceno samo na začetku, ko smo pač morali odgovoriti na postavljeno vprašanje o oceni. Sprememba ocene je dokaz, da je naš projekt voden in kontroliran. Zavedati se moramo, da je napovedovanje prihodnosti operacija, ki temelji na verjetnostnem računu. V trenutku, ko podamo oceno, se zavedamo, da je naša ocena lahko napačna.

Vsa zgoraj navedena dejstva kažejo, da je postavljanje ocen o programskem projektu zelo nedoločen posel, ki nam ne da točno določenih odgovorov. Univerzalne formule, ki bi nam opisala dogajanje na projektu, ni. Zato, da formulo sploh lahko uporabimo, moramo imeti na voljo zadostne količine podatkov. Podatkov pa vedno ni na voljo in v veliko primerih se moramo zadovoljiti s subjektivnimi ocenami ali ocenami, ki temeljijo na intuiciji. Torej, zakaj bi porabili čas in denar, ko pa tudi ocene, ki jih dobimo na bolj formalen način, niso pretirano bolj natančne. Prav gotovo je takšno razmišljanje prisotno v industriji programske opreme. Vendar se podjetja razvijajo in prej kot slej ugotovijo, da je potrebno nekaj narediti tudi na področju postavljanja ocen potrebnega dela na projektu. Vsako stvar se je potrebno naučiti, pri tem uporabiti izkušnje drugih, ki so podobne probleme že reševali in dodati svoje lastne izkušnje, saj le mi sami najbolj poznamo projekt in okolje, v katerem delamo. Nič ni narobe, če so prve ocene napačne, narobe pa je, če jih sploh ne naredimo in se ničesar ne naučimo iz svojih napak.

V nadaljevanju sledi:

- Definicija, kaj cenitev sploh je.
- Sistematičen opis problematike na področju cenitve programskih projektov.

- Sistematičen opis metod, ki nas pripeljejo do ocen.
- Kratek opis evolucije ocenjevanja programskih projektov.
- Opis ciljev naloge.

1.1. Definicija

Cenitev projekta programske opreme je ocena potrebnega dela, denarja in časa, da uspešno končamo projekt. Cenitev je odvisna od razumevanja zahtev projekta, načina dela na projektu, izkušenosti ekipe, ... Ocenjevalec mora razmere na projektu dobro poznati. Opisati mora, na kakšen način je prišel do svojih ocen in kakšna je stopnja zaupanja v oceno. K oceni pa je potrebno dodati tudi seznam faktorjev tveganja, ki lahko ogrozijo izdelano napoved.

Končni cilj ocenjevanja oziroma napovedovanja prihodnosti projekta sta dve oceni, ena je potrebno delo – koliko ur bo ena oseba porabila za dokončanje projekta; drugi je trajanje projekta – koliko koledarskih dni bomo porabili za dokončanje projekta.

Ocena je napoved, ki nam mora povedati, katera vrednost je najbolj verjetna, oziroma katera vrednost je tista, ki bo v končni fazi enako verjetno nižja ali višja od dejanskega rezultata (Tom DeMarco [11]).

Iščemo torej napoved, pri kateri bo končni rezultat z verjetnostjo 0.5 višji, oziroma nižji od naše napovedi.

1.2. Problemi pri cenitvah programskih projektov

Kot smo že ugotovili, je cenitev programskih projektov zahtevna naloga, ki ne pozna univerzalnih odgovorov. Eksperti, ki se ukvarjajo s cenitvami programskih projektov, so razvili številne metode, kako priti do čim bolj realnih rezultatov. Kljub temu, pa se še vedno postavlja vprašanje ali govorimo o znanosti ali umetnosti? Zakaj je cenitev tako zahteven posel, za katerega se izkaže, da največkrat ni zadovoljivo opravljen, tako da načrtovani projekti prekoračijo začetna pričakovanja. Poglejmo si

nekaj osnovnih problemov, s katerimi se srečujemo pri ocenjevanju programskih projektov.

1.2.1. Cilji projekta niso vedno skladni

Ena izmed težav, ki se pojavlja, je neskladnost ciljev, ki so definirani na projektu [1]. Včasih moramo pri izvajanju projekta zadostiti nasprotne interese. Zahteva se, da izvedemo projekt, ki vsebuje določeno funkcionalnost in ima ustrezno zmogljivost, v določenem času in z omejenim stroški. To pa seveda ni mogoče, saj so nam postavili omejitve v vseh treh dimenzijah: število inženirjev, funkcionalnost in čas. Takšne zahteve gotovo onemogočajo proces postavljanja ocen. V tem primeru ponavadi ne postavljamo ocene, ampak izdelujemo načrt, kako ugoditi čim večjemu številu zahtev, kar pa ni realno. Pri tem že v trenutku definiranja ocen vemo, da so uresničljive le, če bi vse dejavnosti na projektu potekale brezhibno. Ocena ne upošteva nikakršnih faktorjev tveganja, predvidimo, da bomo novo tehnologijo spoznali v nekaj dneh, da so specifikacije produktov, ki so vključeni v naš projekt, brez napak, da tako dela tudi vsa programska oprema, ki jo bomo uporabljali na projektu. Ne upoštevamo možnosti odsotnosti članov v ekipi zaradi bolezni, treningov, konferenc, dopustov... To pa seveda ni pravilna pot postavljanja ocene in projekt se bo kmalu znašel v situaciji, ko bo potrebno nekatere od ciljev prilagoditi nastalim razmeram. Takšen način ocenjevanja nas lahko pripelje v situacijo, ko bo naša ocena premajhna ne samo za nekaj deset odstotkov, ampak nekajkrat.

1.2.2. Premalo natančen opis projekta

Naslednja težava, s katero se moramo soočiti v primeru planiranja, je premalo natančen opis projekta [1]. V trenutku, ko želimo napovedati prihodnost projekta, imamo za to na voljo premalo podatkov, specifikacije še niso jasne. Oceno o projektu rabimo na samem začetku, tako da dobimo zeleno luč za izvedbo projekta. V tem trenutku še ne vemo, kako bomo projekt izvedli. Včasih se ukvarjamo s popolnoma novimi tehnologijami, ki jih ne poznamo, ne vemo, kakšen načrt za izvedbo se bo porodil v glavah razvijalcev v naslednjih mesecih. Natančne specifikacije, ki jih lahko imamo za temelj naše ocene o potrebnem delu za izvedbo, bodo na voljo šele, ko bo projekt že v teku in ko bomo porabili že približno petino proračuna, ki nam je na voljo

za izvedbo. Seveda se moramo zavedati, da je ocenjevanje programskega projekta kontinuiran proces, ki ga izvajamo skozi cel življenjski cikel projekta. Dejansko se prve ocene postavljajo v trenutku, ko še nimamo na voljo zadosti podatkov, da bi bile naše ocene kar najbolj pravilne.

1.2.3. Ponovna uporabe kode

Do velikih razlik v količini potrebnega dela prihaja tudi pri poskusih ponovne uporabe kode [1]. Veliko projektov vsaj v določenem segmentu temelji na že izdelani kodi. Seveda pa cena ponovne uporabe kode ni zanemarljiva. Kodo moramo locirati, razumeti in prilagoditi. Mogoče bomo kodo uporabili na drugi ciljni platformi, ki nam ni tako dobro poznana, kot platforma, na kateri smo kodo razvili. Ocene dela pri ponovni uporabi kode so tako zelo različne. V primeru, da kodo že poznamo, je delo dokaj enostavno, v primeru, da se dela loti nekdo, ki kode še ne pozna, bo potrebno delo mnogo večje. Ponovna uporaba kode tako tudi ni vedno cenovno učinkovita in včasih je bolje modul napisati na novo, kot pa ga ponovno uporabiti.

1.2.4. Hitre spremembe v programski industriji

Način razvoja programske opreme se hitro razvija in spreminja, kar dodatno otežuje ocenjevanje [1]. Porajajo se nova orodja, novi principi razvoja programske opreme, ki zahtevajo tudi drugačen pristop pri ocenjevanju potrebnega dela. Novi principi v industriji programske opreme omogočajo razvoj bolj kvalitetne programske opreme z manjšim številom napak, produkti so bolj modularni, kar omogoča lažje vzdrževanje, nekateri procesi omogočajo, da programsko opremo dostavimo uporabnikom hitreje. Tako danes uporabljamo pri razvoju programske opreme že narejene programske produkte (COTS - Commercial Of The Shelf), aplikacijske generatorje in jezike četrte generacije. Ocenjevanje potrebnega dela temelji predvsem na razumevanju parametrov, ki vplivajo na razvoj programske opreme. Zaradi hitrih sprememb na samem razvoju, pa se parametri, ki vplivajo na razvoj, spreminjajo hitreje, kot se postavijo modeli, ki so podprti z zadosti podatki iz preteklosti.

1.2.5. Pomanjkanje znanja o ocenjevanju programskih projektov

Izkaže se, da projektnim vodjem primanjkuje znanja o ocenjevanju programski projektov. Tom DeMarco [11] na podlagi svojih večletnih izkušenj opisuje, kako večina projektnih vodij meni, da tega znanja nimajo dovolj. Ugotavlja pa, da je vzrok, zakaj tega znanja primanjkuje ta, da projektni vodje posvetijo zelo malo časa ocenjevanju projekta, tako da tudi z večletnimi izkušnjami ne postanejo boljši ocenjevalci. Cenitev programskih projektov je kontinuiran proces, projektni vodje pa ga v večini primerov bolj razumejo kot enkraten dogodek, ki je potreben le na začetku projekta. Dodatno pa ugotavlja, da pri postavljanju ocen, delajo veliko napak. Pravzaprav bi njihovemu načinu postavljanja ocen težko rekli cenitev. Poglejmo nekaj primerov, kakšne nepravilnosti se dogajajo pri postavljanju ocen projekta:

- Projektni vodja se odloči, da nove ocene trenutno še ne rabi, saj jo bo gotovo moral narediti pred koncem projekta. Takrat bo imel na voljo bistveno več informacij in bo tako lažje postavil pravilno »oceno«. Zakaj bi torej izgubljal čas, ko pa mora postoriti še toliko drugih zadev, obenem pa ne bo izgledal resno, če bo dvakrat ali še celo večkrat zapored popravil svoje ocene.

- Projektni vodja se zaveda, da stranka ne bo sprejela več kot 20 odstotno zakasnitev projekta. Kljub temu, da ve, da takšno podaljšanje projekta ni zadostno, se zaradi političnih vzrokov odloči, da bo napovedal še sprejemljivo zakasnitev.

- Projektni vodja sploh ne postavi ocene, temveč poskuša ugotoviti, kaj njegov nadrejeni od njega pričakuje. Gotovo se je projektni vodja tega naučil iz izkušenj, vendar njegov način dela ni pravilen. Na takšen način bo seveda ugodil svojemu nadrejenemu, on bo z njim zadovoljen, kar je seveda tudi pomembno. Novih znanj s področja postavljanja ocene projekta, pa na takšen način zagotovo ni pridobil.

- Projektni vodja slabo razume definicijo, kaj ocena je. Mnogokrat podamo za oceno najbolj optimistično verzijo izvedbe projekta, ki pa se zelo verjetno ne bo zgodila.

1.2.6. Pristranskost pri postavljanju ocen

Ocenjevanje projekta je lahko ogroženo tudi zaradi pristranskosti ocenjevalca [11]. Denimo, da mora projekt oceniti projektni vodja, ki je pod pritiskom svojega nadrejenega in neodvisni ocenjevalec, ki bo nagrajen glede na kvaliteto podane ocene. Gotovo se bosta ti dve oceni pošteno razlikovali, saj bo projektni vodja skušal ugoditi svojemu nadrejenemu in si na ta način »prislužiti nagrado«, ocenjevalec pa bo skušal postaviti čim bolj pravilno oceno, saj bo njegova nagrada splavala po vodi, če bo napaka prevelika. Torej moramo ocenjevalce, če hočemo pravilne ocene, tudi pravilno motivirati.

Do pristranskih ocen prihaja lahko tudi iz drugih razlogov. Človeški možgani si dobro zapomnijo dogodke iz bližnje preteklosti, glede dogodkov iz preteklosti pa si bolj zapomnimo tiste, ki so bili izredno ugodni ali izredno neugodni. Ocene, ki jih podamo na osnovi najboljših oziroma najslabših spominov, pa seveda niso pravilne, temveč pretirane ali podcenjene. Posebej, če postavlja oceno posameznik, ki vedno rad obravnava senzacionalne dogodke iz preteklosti, je naša ocena v nevarnosti.

Pretirano samozavesten posameznik ravno tako lahko postavi pretirano optimistično oceno o potrebnem delu na projektu. Posebno kadar gre za del kode oziroma modul, ki je še posebno zahteven, so oceno bistveno podcenjene. Seveda pa se izkaže kasneje, da problem, ki je bil predhodno označen kot »mala malica«, kasneje predstavlja velike probleme pri izvedbi projekta. Ravno tako, kot pretirano samozavesten, nam lahko pretirano previden ocenjevalec tudi poda napačno oceno, saj jo bo, na osnovi slabih izkušenj iz preteklosti, preveč napihnil.

Včasih se lotimo procesa ocenjevanja tako, da postavimo začetno oceno, potem pa samo še poiščemo primerne podatke, da našo oceno upravičimo. Pri tem gre bolj za zagovarjanje naše ocene, ne pa za nepristransko ocenjevanje. Zavedati se moramo, da moramo našo prvotno oceno vedno prilagoditi na osnovi zaključkov, do katerih smo prišli s pomočjo raziskave, ki naj bi potrdila oziroma ovrgla našo prvotno hipotezo.

1.2.7. Pomanjkanje uporabnih podatkov

Bistvenega pomena za postavljanje pravilnih ocen so seveda tudi podatki o preteklih projektih, ki nam pomagajo postavljati pravilne ocene o prihodnosti [11].

Pomembno je tudi vprašanje, kateri podatki so pravzaprav pomembni in koristni pri napovedovanju projekta. Postaviti je potrebno pravilne metrike, ki nam kvantitativno opišejo obseg, kvaliteto in kompleksnost. Seveda podatki niso vedno na voljo, oziroma niso zbrani na enem mestu in v pravilni obliki, da bi nam lahko hitro in učinkovito pomagali pri postavljanju ocen.

1.3. Tehnike cenitev projektov razvoja programe opreme

Trenutno so nam na voljo različne tehnike ocenjevanja projektov razvoja programske opreme, vse pa se srečujejo s problemom dinamičnega razvoja programskih projektov, ki postajajo ne samo večji, temveč tudi kompleksnejši, kar dodatno otežuje določiti natančno oceno razvoja. Eden izmed glavnih ciljev pri ocenjevanju količine dela na projektu je postaviti model, ki bo zanesljivo opisoval način dela na projektu in obenem napovedal ceno razvoja produkta. Tehnike ocenjevanja, ki so nam trenutno na voljo, delimo na različne skupine, kot so:

- Ocenjevanje s pomočjo eksperta.
- Ocenjevanje s pomočjo tehnike »od zgoraj navzdol«.
- Ocenjevanje s pomočjo tehnike »od spodaj navzgor«.
- Ocenjevanje s pomočjo parametričnega modela.
- Ocenjevanje s pomočjo tehnik učenja.
- Ocenjevanje s pomočjo dinamične tehnike.

1.3.1. Ocenjevanje s pomočjo eksperta

Veliko tehnik cenitev temelji na cenitvi s pomočjo eksperta [12]. Nekatere so popolnoma neformalne in temeljijo na izkušnjah vodje projekta s preteklimi podobnimi projekti. Tako je natančnost teh cenitev odvisna od izkušenj, znanja in objektivnosti cenilca. Ocena je lahko enostavno ugibanje, kot na primer projekt A je porabil 100 inženirskih mesecev za izvedbo. Projekt B pa je prejšnjemu projektu

podoben, mogoče večji za 20 odstotkov, tako da ocenjujemo, da bo potrebnih 120 inženirskih mesecev za izvedbo.

Zgornjo preprosto oceno lahko formaliziramo. Povabimo več ekspertov, da nam ocenijo, koliko dela bo potrebno vložiti, da dokončamo projekt. Pri tem zahtevamo od vsakega tri ocene: pesimistično *(x)*, optimistično *(y)* in najbolj verjetno (*z*). Končna ocena je povprečje, ki ga izračunamo s pomočjo enačbe *(x+y+4z)/6*. Takšen način nam normalizira cenitev enega samega eksperta.

Delphi tehnika uporablja cenitve ekspertov na drugačen način. Eksperti naredijo cenitev vsak zase. Ocene temeljijo na izkušnjah, lahko pa tudi uporabljajo kakšno drugo tehniko cenitve, ki jim ustreza. Na podlagi ocen se izračuna povprečna ocena, ki se predstavi skupini. Eksperti dobijo priložnost, da ponovno pregledajo svoje ocene in jih prilagodijo. Nekateri eksperti se pred ponovnim ocenjevanjem posvetujejo, drugi pa ne. Ena izmed možnosti je tudi, da eksperti anonimne ocene ostalih cenilcev pregledajo, preden ponovno podajo svojo oceno.

Na splošno so ocene ekspertov lahko nepravilne iz različnih razlogov. Ocene ekspertov temeljijo predvsem na izkušnjah in paralelah med preteklimi projekti in projektom, ki ga ocenjujemo. V praksi pa le redko najdemo podobne projekte. V primeru razlik je težko določiti, na kakšen način te razlike vplivajo na projekt. Linearnega modela ne moremo uporabljati. Dva razvijalca na enem modulu ne bosta končala dvakrat hitreje kot eden. Produktivnost razvijalcev je lahko zelo različna. Tudi način, ki ga različne organizacije uporabljajo za načrtovanje in razvoj programske opreme, je različen, tako da zelo težko primerjamo cenitve v dveh različnih podjetjih. Ravno tako se podjetja tudi razvijajo, način dela se spreminja, procesi postajajo bolj skladni, kar vse vpliva na samo izvedbo projekta.

Tako je ocenjevanje s pomočjo eksperta preveč izpostavljeno napakam, ki so rezultat ocenjevalčeve objektivnosti, subjektivnosti ali pa celo napačnega razumevanja problema. Naslednja težava te tehnike je tudi ta, da ne moremo oceniti natančnosti ocene, ki je podana. Tako se te tehnike ne uporabljajo v enaki meri kot formalne tehnike, lahko pa s takšno tehniko zberemo koristne informacije in podatke, ki jih drugi modeli uporabljajo kot potrebne vhodne podatke.

1.3.2. Ocenjevanje s pomočjo tehnike »od zgoraj navzdol«

Pri ocenjevanju projekta s pomočjo tehnike »od zgoraj navzdol« skušamo oceniti celotno ceno projekta, potem pa le-to razbiti na posamezne faze projekta. Tako denimo lahko ocenimo, da bomo porabili za specifikacijo 20% celotnega dela na projektu, za načrtovanje 20%, za izvedbo 30% in za testiranje 30% dela na projektu. Takšen način sklepanja je lahko pravilen le v primeru, ko delamo na projektih, ki so si med seboj zelo podobni, tako da lahko na osnovi preteklih projektov podamo zgornjo oceno. Glede na to, da takšen način dela razbije oceno na manjše dele, bodo te posamezne ocene pravilne samo, če je celotna ocena podana pravilno.

1.3.3. Ocenjevanje s pomočjo tehnike »od spodaj navzgor«

Ocenjevanje s pomočjo tehnike »od spodaj navzdol« je zelo natančen in časovno kompleksen proces. Posameznik poskuša nalogo, za katero je zadolžen, razbiti na več podnalog, te spet na aktivnosti in podaktivnosti, dokler ne pride do hierarhične strukture (WBS - Work Breakdown Structure), ki opisuje, kaj vse je potrebno postoriti, da bo njegova naloga uspešno končana. Posameznik nato poskuša ovrednotiti vsako posamezno komponento. Ocene posameznih komponent se potem akumulirajo in sestavijo v oceno celotne naloge. Pri ocenjevanju posameznega dela poskušamo oceniti, kakšno delo je potrebno za to komponento. Ker so posamezni elementi ocenjevanja dokaj majhni, jih lahko lažje pravilno ovrednotimo. Za vsako komponento ocenimo, koliko dela je potrebno za načrtovanje, izvedbo, testiranje, opis, konfiguriracijo in namestitev.

Ocena dobljena na takšen način je lahko dovolj natančna, ker posameznik ocenjuje delo, ki ga dobro pozna. Ponavadi je ocenjevalec izkušen inženir, ki dobro pozna področje, ki ga ocenjuje in je sposoben svojo oceno tudi utemeljiti in razložiti. Seveda tudi pri takšnih ocenah prihaja do napak. Če so vse ocene nekoliko pretirane, potem s seštevanjem le-teh dobimo pretirano skupno oceno. Ravno tako velja obratno, če so vse ocene podcenjene, s seštevanjem le-teh dobimo podcenjeno oceno. Torej je natančnost modela odvisna od natančnosti posameznih ocen.

Po drugi strani ima takšno ocenjevanje lahko tudi svoje slabosti. Za natančno podano oceno moramo projekt dovolj dobro razčleniti, kar zahteva ogromno dela in seveda tudi dovolj globoko poznavanje projekta in njegovih načrtov. Zato je takšno oceno zelo težko podati na začetku projekta. Ponavadi se takšen način uporablja pri faznem ocenjevanje projekta, kjer skušamo skozi ves življenjski cikel projekta na koncu vsake faze oceniti, koliko dela nas čaka v naslednji fazi. Pri takšnem načinu ocenjevanja se sprijaznimo, da je nesmiselno podati oceno za celoten projekt, ker je le ta premalo natančna. Odločimo se, da bomo vsako fazo ocenili posebej, torej bomo vsako fazo obravnavali kot samostojen projekt. Za vsako fazo določimo, katere kriterije moramo izpolniti, da lahko opravimo prehod v naslednjo fazo.

1.3.4. Ocenjevanje s pomočjo parametričnega modela

Razširjen način ocenjevanja potrebnega dela za dokončanje programskih projektov je tudi uporaba parametričnih modelov [19]. Pravilno postavljen parametrični model nam omogoči bolj učinkovito postavljanje ocen na začetku projekta in pa tudi izboljšanje natančnosti, kvalitete in skladnosti ocen. S pridobljenimi ocenami si lahko pomagamo tudi pri ocenah tveganja, planiranju in kontroliranju projekta. S takšnim načinom dela izboljšamo tudi interne procese v podjetju, podjetje postaja zrelejše in bolj izkušeno.

Metode, ki temeljijo na parametričnem modelu, poskušajo čim bolje oceniti statistično relacijo med odvisno spremenljivko ceno in ostalimi spremenljivkami, ki vplivajo na razvoj produkta, kot so velikost projekta, kompleksnost projekta, tip projekta, struktura ekipe, izkušenost ekipe, ciljna platforma, ... Parametrični modeli, ki so na voljo širokemu krogu uporabnikov na trgu, so komercialni modeli. Podjetja pa lahko razvijejo svoje lastne parametrične modele.

Komercialni modeli so splošni modeli, ki so zgrajeni na osnovi podatkov dobljenih iz različnih podjetji, ki se ukvarjajo s programskimi projekti. Preden jih začnemo uporabljati, je potrebno takšne modele kalibrirati, torej prilagoditi podjetju, ki ga bo uporabljal. Kalibracija nam omogoča, da prilagodimo splošne parametre modela lokalnim razmeram v podjetju, ki bo model uporabljal. Kalibriramo torej tako, da izračunamo prilagoditveni faktor, ki bo odpravil razliko med splošno napovedjo in napovedjo posebej narejeno za podjetje, ki ga uporablja. Potem, ko smo izvedli

potrebno kalibracijo modela, je samo ocenjevanje s pomočjo parametričnega modela hitro in enostavno. Seveda, če pravilno razumemo model in njegove ključne parametre. Na trgu je na voljo kar precej modelov za parametrično cenitev, kot so na primer SLIM, PRICE-S, SEER-SEM, COCOMO...

Za izgradnjo internega parametričnega modela se odločimo, kadar nam splošni modeli na zadostujejo in vidimo da potrebujemo svoj model zato, da zadostimo našim ključnim potrebam ocenjevanja. Nekatera podjetja pridejo do lastnega parametričnega modela tudi nekoliko drugače. Na začetku skušajo s pomočjo analize podatkov, ki so jim na voljo, priti do nekih preprostih zakonitosti, ki vladajo na projektih. Tako skušajo določiti razmerja med neko neodvisno in odvisno spremenljivko. Ena izmed takšnih zakonitosti bi lahko bila razmerje med časom, porabljenim za načrtovanje projekta in časom porabljenim za kodiranje načrtovanega projekta. Več takšnih ugotovljenih razmerij, lahko povežemo med seboj in v končni fazi nas ta povezava pripelje do lastnega parametričnega modela. Interni modeli nastanejo na podatkih, ki so bili zbrani v podjetju samem, zato kalibracija teh modelov ni potrebna.

Modeli nam torej poskušajo z matematičnimi izrazi opisati dogajanje na projektu. Kaj pa je dejansko tisto, kar najbolj vpliva na izvedbo projekta? Večina modelov ocenjuje, da je velikost projekta najbolj pomembna neodvisna spremenljivka, ki vpliva na količino potrebnega dela za dokončanje projekta. Modeli opisujejo velikost projekta s številom vrstic kode, kar seveda ni podatek, ki bi nam bil na voljo že na samem začetku projekta. Tako smo prevedli problem ocene dela na projektu v problem ocene števila vrstic kode. Število vrstic kode, je seveda odvisno tudi od programskega jezika, ki ga uporabljamo.

Razvijalci modelov so skušali ta problem odpraviti tako, da so razvili tudi metode, ki nam omogočajo oceniti velikost projekta. Poznamo na primer oceno s pomočjo funkcionalnih točk, objektnih točk, ... Funkcionalne točke so utežena vsota petih različnih faktorjev, ki opisujejo uporabnikove zahteve na projektu. To so programski vhodi, izhodi, logične datoteke, povpraševanje in vmesniki. To metriko lahko določimo že bistveno prej v življenjskem ciklu projekta.

Poleg velikosti, ki je najpomembnejši parameter za določitev potrebnega dela na projektu, so za pravilno oceno potrebni tudi drugi parametri. Ti parametri se razlikujejo med modeli, vendar jih lahko na splošno razvrstimo v naslednje štiri skupine:

- Parametri programske opreme - kompleksnost, programski jezik, uporaba že obstoječe kode, zahtevana zanesljivost.
- Parametri strojne opreme – omejitve resursov, stabilnost platforme, ….
- Parametri projektne ekipe – sposobnost ekipe, izkušnje ekipe, ….
- Parametri projekta – orodja in tehnike, razpršenost ekipe, časovne omejitve, ….

Pri ocenjevanju teh parametrov mora ocenjevalec sodelovati z izkušenimi člani ekipe, ki poznajo tehnično ozadje projekta in pa tudi način dela skupine. Le na takšen način lahko pridemo do pravilnih ocen parametrov in posledično tudi do pravilnih ocen potrebnega dela na projektu. Zavedati se moramo dejstva, da modeli niso čudežne skrinjice, ki nam bodo podale natančno oceno. Ocena je še vedno v veliko meri odvisna od ocenjevalca, njegovega razumevanja modela in okolja, ki ga ocenjuje. Model nam ne bo dal zadovoljivih rezultatov, če ga ne bomo pravilno kalibrirali. Kalibracija ne bo uspela, če nimamo na voljo konsistentnih podatkov o naših preteklih projektih.

S parametričnimi modeli lahko na hiter in učinkovit način dobimo oceno projekta, seveda, če jih pravilno razumemo. Ker pa ti modeli temeljijo na podatkih o preteklih projektih, so njihova glavna pomanjkljivost nepredvidljive situacije.

1.3.5. Ocenjevanje s pomočjo tehnik učenja

Med tehnike učenja sodijo tako stare ne-avtomatizirane tehnike, kot tudi novejše avtomatizirane tehnike. Med stare tehnike sodijo študije primerov (Case Studies) [15]. Študije primerov predstavljajo induktiven proces, kjer ocenjevalci in planerji skušajo z raziskovanjem specifičnih primerov priti do splošnih pravil. Natančno analizirajo projekte, tako pogoje in omejitve okolja, tehnične in upravljavske odločitve, ter končne pozitivne oziroma negativne rezultate. Iz teh analiz poskušajo konstruirati povezave med vzrokom in posledico. Skušajo poiskati primere, kjer so projekti podobni projektu, za katerega skušajo podati oceno, pri tem pa uporabljajo pravilo analogije. Podobni projekti naj bi zahtevali enako mero dela in časa. Projekti, na katerih je narejena študija primerov, so lahko tako interni, kakor tudi eksterni.

Interni projekti so načeloma boljši izvor podatkov za takšen način ocenjevanja. Dobro dokumentirani eksterni projekti pa so lahko ravno tako koristni.

Nevronske mreže pa so primer novejše avtomatizirane tehnike, kjer poskušamo zgraditi modeli, ki se »uči« iz prejšnjih izkušenj [15]. Ocenjevalni model lahko »naučimo«, da prilagodi parametre algoritma tako, da se zmanjša razlika med poznanimi dejstvi in napovedmi modela.

Nevronska mreža predstavlja večje število med seboj povezanih odvisnih enot. Tako lahko te enote uporabimo kot predstavnike različnih aktivnosti, ki so vključene v programski projekt. V nevronski mreži vsake enota imenovana nevron predstavlja neko aktivnost. Vsaka aktivnost ima svoj vhod in izhod. Znotraj vsake enote se opravi izračun, ki na osnovi vhodnih parametrov izračuna uteženo vsoto. Če le ta preseže določeno vrednost, enota da vrednost na izhod. Izhod postane vhod drugim enotam, dokler ne pridemo do končne izhodne vrednosti, ki da napoved za celotno mrežo, torej izračuna potrebno delo za dokončanje projekta.

Nevronsko mrežo razvijemo tako, da jo »učimo« na podatkih preteklih projektov. Podamo dejanske podatke in s pomočjo algoritmov, ki prenašajo informacijo naprej ali nazaj, »naučimo« mrežo, kakšni vzorci so se pojavljali v teh podatkih. Če na primer podatki o preteklih projektih vsebujejo informacijo o izkušenosti ekipe, nam nevronska mreža lahko poišče informacijo o povezavi med izkušenostjo ekipe in količino potrebnega dela, da se projekt konča.

Kar nekaj raziskovalcev je poskušalo razviti model za cenitev s pomočjo nevronske mreže. Nekatere napovedi so bile celo boljše kot napovedi narejene s pomočjo parametričnih modelov kot so COCOMO in SLIM. Kljub temu pa ima takšen način ocenjevanja tudi svoje pomanjkljivosti. Tako je natančnost modela odvisna od topologije nevronske mreže in pa tudi začetnih naključnih uteži, ki se določijo nevronom. Nevronske mreže morajo imeti zadovoljivo velik vzorec, če želimo, da nam bodo dale dovolj dobre rezultate. Takšne podatke pa je včasih težko dobiti, tako nam prav maloštevilnost podatkov, ki so nam na voljo, ogroža uporabnost tehnike.

1.3.6. Ocenjevanje s pomočjo dinamične tehnike

Tehnika poskuša podpreti spoznanje, da se delo, potrebno na programskem projektu, in parametri, ki vplivajo na oceno, spreminjajo med samim projektom [15].

Gre za pomembno razliko od ostalih tehnik, ki skušajo na osnovi statističnih modelov in posnetkov narejenih v neki določeni situaciji napovedati oceno. Faktorji kot so roki, ekipa, specifikacije, potrebe po treningih, pa se spreminjajo skozi življenjski cikel projekta, kar seveda povzroči tudi različno produktivnost projektne ekipe.

Najbolj znana dinamična tehnika je bila razvita na osnovi dinamičnega modeliranja sistema, razvila pa jo je Jay Forrester pred štiridesetimi leti. Gre za simulacijsko tehniko, kjer so rezultati modela in njegovo obnašanje prikazani kot grafi, ki se spreminjajo s časom. Modeli so prikazani kot mreže, ki se spreminjajo s pozitivnimi oziroma negativnimi povratnimi zankami. Elementi v modelu so izraženi kot dinamično spreminjajoči nivoji – vozli, pretoki med nivoji – povezave med nivoji in informacijo o sistemu, ki se spreminja s časom in avtomatično vpliva na nivo pretoka med nivoji – povratne zanke.

S pomočjo takšnega modela je Madachy razložil slavni Brooksov zakon, ki pravi, da dodajanje ljudi na projekt, ki kasni, še dodatno zakasni projekt. Novi ljudje niso takoj usposobljeni za delo, potrebujejo trening, pri tem dodatno obremenjujejo že obstoječo ekipo. Poleg tega je potrebno v večji ekipi tudi več medsebojne komunikacije, kar pomeni, da manj časa ostaja za delo na projektu.

1.3.7. Povzetek

Pregledali smo različne možne tehnike, ki se lahko uporabljajo pri ocenjevanju potrebnega dela na programskih projektih. Vse te tehnike imajo svoje prednosti in slabosti. Ne moremo pa jih deliti na dobre in slabe. Pomembno je, da se zavedamo zakaj prihaja do razlik pri ocenah in znamo te vzroke ustrezno ovrednotiti in razložiti. Pri samem ocenjevanju projekta, je vedno smiselno uporabiti dve različni tehniki. To nam omogoča, da primerjamo dobljene rezultate, skušamo razložiti nastale razlike in na ta način še bolj poglobljeno razumemo problem, oziroma projekt.

1.4. Opis evolucije tehnik ocenjevanja

Zanimiva je tudi sama evolucija razvoja tehnik ocenjevanja, ki jo je opisal Stuzke [1].

1.4.1. Šestdeseta leta

Začetki cenitev programske opreme segajo v leto 1960, ko je Frank Freiman razvil koncept parametričnega ocenjevanja, kar je vodilo v razvoj modela PRICE, ki je bil v začetku namenjen ocenjevanju razvoja strojne opreme. To je bilo prvo računalniško orodje namenjeno ocenjevanju potrebnega dela, ki je bilo v sedemdesetih letih prilagojeno tudi ocenjevanju dela na programskih projektih.

1.4.2. Sedemdeseta leta

Sedemdeseta leta so bila dokaj aktivna na področju postavljanja modelov, za določanje potrebnega dela na projektu. V tem času je postalo izredno pomembno ocenjevanje dela na projektih, tako da je bilo tudi raziskovalno delo na tem področju zelo živahno. Projekti so postajali vse večji, mnogi od njih pa so končali v rdečih številkah. Frederic Brooks je opisal probleme, ki jih je izkusil kot projektni vodja pri razvoju operacijskega sistema v IBM-u v svoji knjigi The Mythical Man Month [14]. Knjiga na duhovit in realističen način opisuje težave, s katerimi so se spopadali. Zanimivo dejstvo je, da je knjiga po skoraj tridesetih letih še vedno aktualna.

V teh letih se je največ razvijalo v jezikih kot so FORTRAN, ALGOL, JOVIAL in Pascal. Programska orodja so bila omejena le na urejevalnike besedil in prevajalnike. Sistemi so bili ponavadi grajeni od začetka, tako da so tehnike ocenjevanje predvsem pokrivale področje novega razvoja.

V tem času so številni avtorji analizirali podatke o projektih in ugotavljali, kateri faktorji vplivajo na ceno razvoja programske opreme. Pomembni faktorji so se določevali predvsem s korelacijskimi tehnikami, v sam model pa so se vključili z regresijskimi tehnikami. Regresija je statistična metoda za napovedovanje vrednosti ene ali več odvisnih spremenljivk iz zbirke neodvisnih spremenljivk. Koeficienti modela so določeni, tako da določijo najboljše možno ujemanje zbranim podatkom. Pri tem se ponavadi srečujemo s problemi, saj nam primanjkuje podatkov, obenem pa tudi v razvoju programske opreme ne moremo definirati nobenih zakonov, ki bi omejili možno postavitev modela.

Prototip parametričnih modelov je gotovo COCOMO, katerega avtor je Barry Boehm. Prvič je bil opisan v knjigi Software Engineering Economics [10]. Različne

verzije produkta COCOMO se še vedno uporabljajo, saj je produkt precej razširjen. COCOMO nam nudi formule za oceno dela in časa potrebnega za dokončanje projekta. Nominalno delo je predvsem odvisno od velikosti projekta, ki se meri v številu vrstic kode. Izračun prilagodimo potem še z upoštevanjem faktorjev, ki opisujejo atribute projekta, produkta, osebja in opreme. Prilagojen izračun dela na koncu uporabimo za izračun potrebnega koledarskega časa za dokončanje projekta. COCOMO nam nudi tudi različne načine dela, izberemo pa lahko tudi, kako podroben opis projekta smo pripravljeni podati, da dobimo oceno.

Naslednji parametrični model za ocenjevanje dela, ki je bil razvit v tem obdobju, je PRICE-S. PRICE-S sta razvila od 1975 leta do 1977 leta Frank Freiman in Robert Park. Model je temeljil na podatkih, zbranih na več kot 400 projektih. Ti parametrični modeli so bili prvi avtomatizirani in splošno dostopni. Program, ki ga je razvil William Rapp, je tekel na mainframe računalniku. Tudi PRICE-S najprej izračuna delo kot odvisno spremenljivko velikosti projekta, ki je izražena v vrsticah kode, programskega jezika in kompleksnosti aplikacije. Nominalno delo je potem prilagojeno s faktorjem produktivnosti. Faktor produktivnosti je določen glede na tip jezika, kompleksnost aplikacije in tip platforme. Prav tako lahko faktor produktivnosti določimo tudi sami s pomočjo kalibracije.

Pomanjkljivost teh modelov je, da je neodvisna spremenljivka, ki določa potrebno delo, velikost projekta, pravzaprav odvisna spremenljivka, ki ni na voljo na samem začetku projekta v trenutku, ko želimo podati oceno. Tako sta konec sedemdesetih let Allan Albrecht in John Gaffney razvila analizo funkcionalnih točk FPA (Function Point Analysis), ki nam omogoča določiti velikosti projekta in posledično tudi potrebnega dela na projektu. Komponente sistema sta klasificirala v pet različnih tipov (vhod, izhod, povpraševanje, logična interna datoteka in eksterna datoteka, ki se uporablja kot vmesnik). Vsak tip ima določeno ceno, ki je sorazmerna delu, ki je potrebno, da ga razvijemo. Tako ocenimo velikost projekta kot vsoto vseh definiranih funkcionalnih točk.

V sedemdesetih letih je Lawrance H. Putnam razvil SLIM (Software Life Cycle Model) na osnovi podatkov 50 programskih projektov v ameriški vojski. Ugotovil je, da je delo, potrebno za dokončanje projekta, sorazmerno kubu velikosti projekta in obratno sorazmerno času razvoja na četrto potenco. Druga enačba, ki jo je definiral pravi, da je delo sorazmerno kubu času razvoja. Če rešimo ti dve enačbi, nam rešitev predstavlja minimalni čas za razvoj projekta.

1.4.3. Osemdeseta leta

V osemdesetih letih se je razširila uporaba osebnih računalnikov. Modeli, ki so bili definirani v sedemdesetih letih so bili implementirani. Razvila se je Ada, namenjena zmanjšanju stroškov pri implementaciji velikih sistemov. Tako je ekipa, ki jo vodi Barry Boehm, razvila model Ada COCOMO.

Robert C. Tausworthe je razširil delo Boehma, Herda, Putnama, Walsona in Felixa in razvil model za NASO, ki ga je kasneje razširil še Donald Reifer, ki je razvil model SOFTCOST-R.

Randall W. Jansen je nadaljeval Putnamovo delo. Izločil je nekatere nezaželene učinke modela SLIM. Jensen je postavil model, v katerem je delo sorazmerno kvadratu velikosti projekta in obratno sorazmerno kvadratu trajanja projekta. Daniel Galorat je nadaljeval Jensenovo delo in razvil model poznan kot SEM (Software Estimation Model), ki je poznan kot del SEER modela (System Evaluation and Estimation of Resources). Verzija SEER-SEM modela 4.5 je prišla na trg leta 1996.

Albrecht je leta 1984 objavil razširjen model funkcijskih točk. Nova metoda deli komponente določenega tipa glede na njihovo kompleksnost in temu ustrezno določi pripadajočo težo. Komponento ocenimo ali ima nizko, srednjo ali visoko kompleksnost. Ta metoda je osnova standarda IFPUG (International Function Point Users Group).

Modela funkcijskih točk razširil je še Capers Jones, ki je vključil v analizo še kompleksnost algoritmov. Njegova metoda, ki jo je imenoval Feature Point Method, je dodala še šesto dimenzijo - algoritme. Pri svoji metodi ni upošteval klasifikacije kategorij po kompleksnosti.

Charles Symons je predlagal še eno spremembo pri določanju FPA. Skušal je zmanjšati vpliv na model v primeru, če delamo z datotekami, ali če govorimo le o enem sistemu. Njegova metoda, imenovana Mark II Functional Points temelji na štetju logičnih transakcij.

1.4.4. Devetdeseta leta

V devetdesetih so se procesi razvoja programske opreme spet močno spreminjali, zato so se tudi na področju definicije modela za oceno potrebnega dela dogajale spremembe. Barry Boehm je razvil nov model COCOMO imenovan COCOMO II. COCOMO II omogoča, da v kasnejših fazah, ko imamo na voljo več informacij, te tudi uporabimo pri procesu ocenjevanja. Omogoča nelinearno oceno ponovne uporabe programske opreme, ocenjevanje dela potrebnega na podpori produkta, daje podporo različnim tipom življenjskega cikla projekta ...

Scott Whitmire je razširil tudi FPA, ki je sedaj podprt tudi za znanstvene sisteme in sisteme, ki delujejo v realnem času. Svoj model je definiral na osnovi trditve, ki jo je postavil Tom DeMarco, da je vsa programska oprema skupek treh komponent: podatkov, funkcij in kontrol.

John Gaffney je v eni izmed svojih raziskav ugotovil, da je funkcijska analiza prav tako natančna, če upoštevamo samo vhode in izhode in zanemarimo ostale tri komponente.

1.4.5. Kako naprej?

Gotovo na tem področju še nismo zakrožili celotne zgodbe. Način razvoja programske opreme se spreminja, tako da bo potrebno temu razvoju prilagoditi tudi modele. Vsaka organizacija, ki želi napredovati na tem področju, mora tem modelom slediti, obenem pa zelo dobro razumeti procese, po katerih delajo in kako ti procesi vplivajo na samo postavitev modela. V določenih primerih je mogoče tudi boljše, če razvijemo lasten parametričen model, v primeru, da so procesi bistveno drugačni od predpostavljenih.

Seveda je za uspeh teh modelov pomembna tudi pomoč s strani vodstva organizacije, ki mora spodbujati zbiranje podatkov na vseh nivojih, ki so pomembni za uspešno definicijo modela ali kalibriranje obstoječih modelov. Pomembna je tudi standardizacija podatkov, tako da so podatki med seboj primerljivi. Podjetje mora postaviti infrastrukturo, ki je temelj njihovih procesov in ki omogoča uporabo tudi bolj formalnih tehnik s področja ocenjevanja dela na programskih projektih.

1.5. Cilji naloge

Sama cenitev programskih projektov je izredno zahtevna naloga, zato se mi je zdela ta tema zanimiva za magistrsko nalogo. Na razvoju programskih projektov delam že deset let, tako da imam s to temo že nekaj izkušenj. V vseh teh letih sem se pojavljala v različnih vlogah ocenjevalca in seveda pri tem nabirala dragocene izkušnje. Kljub temu, da sem ves čas delala na istem produktu, pa se je način dela v teh desetih letih izredno spremenil, tako kot se je spreminjal tudi način dela na projektih. Kako poteka cenitev programskega projekta sedaj, ko na projektu sodeluje tudi več kot 70 ljudi, na kakšen način pridemo do ocene velikosti in zahtevnosti nove verzije produkta? Gre za kompleksen način ocenjevanja, v katerem je udeleženih precej ljudi, ki sodelujejo na tem projektu. V nalogi bom opisala, kako takšen postopek ocenjevanja poteka in kakšna je vloga posameznikov, ki so vpleteni v ta proces.

Drug cilj naloge je bil spoznati enega od parametričnih modelov, ki se trenutno uporabljajo. Izbrala sem parametričen model COCOMO II in ga opisala v enem izmed poglavji. Za ta model sem se odločila, ker je enostavno dostopen in ker podpira tudi novejše principe, ki se uporabljajo v razvoju programske opreme.

Pomemben cilj naloge je bil poizkus uvedbe parametričnega načina ocenjevanja, ki služi kot vzporedno orodje, s katerim si lahko pomagamo priti do ustreznih ocen o ceni projekta. Model sem prilagodila za uporabo na projektih Data Protector. Primer ocene pa je bil narejen za trenutni projekt, ki je aktiven – Aragon. Kot osnovo za parametričen način ocenjevanja dela sem uporabila parametričen model COCOMO II.

Pomembna komponenta pri razvoju parametričnega modela so bili tudi zgodovinski podatki o preteklih projektih na razvoju programskega produkta Data Protector. Nekaj teh podatkov je bilo sicer že na voljo, vendar sem večino pomembnih podatkov še enkrat zbrala, ker sem želela delati s konsistentnimi podatki. Posledično sem definirala tudi predlog, kakšna naj bi bila struktura arhiva Data Protector, ki bi se uporabljala kot osnova pri različnih primerjavah med projekti in kot osnova za parametrično ocenjevanje.

Če še enkrat povzamem, so bili cilji naloge naslednji:

- Predstavitev načina ocenjevanja potrebnega dela na programskih projektih Data Protector (Cenitev projektov Data Protector).

- Podrobna predstavitev parametričnega modela COCOMO II, ki je bil vzet kot osnova za parametričen način cenitve projekta (COCOMO II).
- Kalibracija modela COCOMO II na Data Protector projekte zbrane v arhivu in cenitev tekočega projekta s pomočjo modela COCOMO II. Prilagoditev modela COCOMO II za tekoči Data Protector projekt (Uporaba modela COCOMO II na projektu Aragon).
- Definicija strukture arhiva Data Protector, ter zbiranje podatkov o zadnjih petih projektih (Struktura arhiva Data Protector).

2. Cenitev projektov Data Protector

Cilj tega poglavja je opisati, kako poteka način ocenjevanja potrebnega dela na projektih razvoja produkta Data Protector. Data Protector je produkt, ki je v lasti podjetja Hewlett Packard in nudi možnost varnostnega shranjevanja podatkov. Produkt se je v veliki večini razvil v našem podjetju v zadnjih desetih letih. Razumevanje procesov, ki se odvijajo v okviru določanja cene projekta, je v veliki meri odvisno tudi od same organizacije ekipe, lastnosti produkta, ki ga razvijamo, stranke, končnih strank, zato sledi v nadaljevanju kratek opis teh tipičnih lastnosti projektov.

2.1. Lastnosti Data Protector projektov

Tipično so projekti na razvoju produkta Data Protector veliki projekti na katerih sodeluje več deset ljudi in katerih velikost obsega potrebno delo nekaj 100 EM (Engineering Month). V času, ko na projektu dela največ ljudi, je lahko na projektu aktivnih tudi preko 70 inženirjev.

Produkt je last stranke Hewlett Packard, končni uporabniki tega produkta pa so podjetja, ki uporabljajo ta produkt kot osnovo za varnostno shranjevanje podatkov. Na produktu delata dve ekipi, ekipa iz podjetja Hermes Softlab in ekipa iz podjetja Hewlett Packard. Ekipa na strani stranke je pri razvoju produkta odgovorna za marketing, prodajo, tehnični marketing in kontrolo samega razvoja, včasih pa tudi za del razvoja.

Razvoj in vzdrževanje produkta je odgovornost naše projektne ekipe. Aktivnosti, ki so vključene v razvoj projekta so: raziskava zahtev, definicija specifikacij, kodiranje, pisanje tehnične in uporabniške dokumentacije in testiranje. Vse te aktivnosti so razdeljene v 5 faz: faza definicije zahtev projekta, faza definicije specifikacije projekta, faza implementacije projekta, faza prevzema projekta in faza v okviru katere formalno zapremo projekt. Model, ki se uporablja za razvoj teh projektov je linearen model življenjskega cikla projekta. Bolj natančno je življenjski cikel projekta razložen v poglavju 2.

Ponavadi so projekti časovno omejeni, torej čas predstavlja eno glavnih omejitev na projektu. Stranka določi do katerega časovnega termina se mora pripraviti nova množica funkcionalnosti za trg. Torej je eden izmed glavnih ciljev projekta časovni okvir, znotraj katerega mora priti funkcionalnost, ki je izbrana za ta projekt, do končnih strank.

Tim je organiziran matrično. Inženirji, ki pripadajo določeni skupini, delajo na različnih Data Protector projektih, ki tečejo paralelno. Paralelni projekti so pomembni iz več razlogov, predvsem pa bi izpostavila dva. Vsi inženirji niso ves čas popolnoma zasedeni na enem projektu. Predvsem pa pride do razlik v fazi 1, ko se ukvarjamo z zahtevami in na tem predvsem sodeluje del ekipe izkušenih razvijalcev in v fazi 4, ko je v teku sistemsko testiranje in testiranje v okviru prevzema produkta, kjer se največ dela vloži v samo testiranje produkta. Zato se ponavadi nov projekt štarta v obdobju, ko je tekoči projekt v fazi sistemskega testiranja. Drugi razlog za paralelne projekte pa so zahteve, da v okviru Data Protector produkta podpremo nove tehnologije, obenem pa se tudi zavedamo, da sama tehnologija še ni popolnoma zgrajena in definirana. V takšni situaciji rajši štartamo majhen, vendar tehnološko zahteven projekt. Ko pripeljemo projekt do konca faze 3, pa ga vključimo v enega od glavnih projektov.

Znotraj vsakega projekta inženirji in skupine poskrbijo za segment, za katerega so zadolženi. Skupina, ki dela na programu Data Protector je razdeljena na:

1. Vodstveno ekipo (MGMT - Managment), ki je zadolžena za vodenje skupin in projektov in pa tudi eksterno komunikacijo s stranko, oziroma projektnim vodjem na strani stranke.

2. Ekipa razvijalcev (DEV - Development), ki je zadolžena za razvoj produkta. Ta je sestavljena iz več podtimov, vsak od njih je zadolžen za enega ali več modulov produkta Data Protector. Člani razvojne ekipe so zadolženi za pregled zahtev, pisanje specifikacij, kodiranje, testiranje enot, tehnično dokumentacijo, pregled testnih planov in planov uporabniške dokumentacije.

3. Ekipa za uporabniško dokumentacijo (LP – Learning Products), ki je zadolžena za pisanje uporabniške dokumentacije.

4. Ekipa za upravljanje konfiguracije (CM – Configuration Management), ki poskrbi za ustrezno prevajanje in pakiranje produkta in ki vzdržuje potrebno infrastrukturo razvojnega okolja.

5. Testna ekipa (QA – Quality Assurance), ki je zadolžena za testiranje in zagotavljanje ustrezne kvalitete na projektu.

6. Ekipa, ki dela na podpori produkta Data Protector (CPE – Customer Product Experience), ki je zadolžena za vzdrževanje produkta potem, ko je že na trgu. CPE ekipa je organizacijsko ločena od skupin, ki delajo direktno na razvoju, je pa tudi vključena v procese, predvsem v smislu informiranja, kaj v novem produktu bo. Sodeluje pa tudi na pregledih eksterne specifikacije in v času sistemskega testiranja preveri produkt. Preverja tako nove funkcionalnosti, kot tudi stare, če se jim zdi, da je katero od področji bolj šibko.

Ponavadi gre pri razvoju nove verzije produkta za razvoj izbrane množice novih funkcionalnosti. Več o sami klasifikaciji novih funkcionalnosti je razloženo v poglavju 4.1.1. Tu pa bi se samo dotaknila posebne vloge lastnika funkcionalnosti. Ker se znotraj novega projekta razvija več 10 novih funkcionalnosti, se za vsako od teh določi lastnik, ki poskrbi, da se na vseh potrebnih segmentih funkcionalnost ustrezno doda v produkt. Tipično je lastnik funkcionalnosti član razvojne ekipe, ki poskrbi tako za pregled zahteve, pisanje specifikacije, implementacijo ali le pregled implementacije, pregled testnega plana, pregled uporabniške dokumentacije. Ko so vse aktivnosti, ki

so potrebne za to, da se funkcionalnost doda v produkt dokončane, je on tisti, ki odloči, da se funkcionalnost tudi formalno vključi v produkt.

2.2. Planiranje projekta

Planiranje se začne takoj na samem začetku projekta in je ena najpomembnejših aktivnosti prvih dveh faz projekta. Seveda se planiranje ne konča, potem ko imamo pripravljen natančen plan, temveč ga nadaljujemo tudi v naslednjih fazah projekta. Cilj prve faze je, da razumemo zahteve in da nam je jasen obseg projekta. Glavni cilji druge faze pa so: definicija eksterne specifikacije nove verzije produkta, definicija načrta produkta in pa natančen plan projekta. Še enkrat poudarimo, da se planiranje ne konča ob koncu prvih dveh faz projekta, temveč se nadaljuje tudi v naslednjih fazah. Glavna aktivnost naslednjih faz, kar še tiče planiranja, je spremljanje tekočih aktivnosti projekta in seveda ponovno planiranje, če ugotovimo, da katera od teh aktivnosti ni po trenutnem planu in zamude ni mogoče zaradi objektivnih vzrokov nadoknaditi kako drugače.

Rezultati prvih dveh faz, ki so povezani s planiranjem so naslednji:

1. Plan kvalitete, v katerem na formalen način definiramo, kaj so izhodi posameznih faz projekta. Lastnik tega plana je projektni vodja.

2. Tabela, ki določa obseg projekta. V tej tabeli za vsako od novih funkcionalnosti določimo lastnika, skupino, ki ji pripada, nivo potrebne spremembe za funkcionalnost in vpliv na module produkta. Lastnik tabele je projektni vodja.

3. Grobi plan, ki ga postavimo v okviru prve faze. S pomočjo tega plana skušamo določiti okvirni plan projekta in na ta način preprečiti, da bi izgubljali preveč časa na funkcionalnostih, ki jih na koncu ne bomo mogli vključiti v projekt. Lastnik plana je projektni vodja.

4. Plan dela, kjer združimo pripravljene plane posameznih vodij timov, ki s pomočjo metode ocenjevanja »od spodaj nazvgor« razčlenijo potrebno delo za dodajanje novih funkcionalnosti, katerih lastniki so člani njihovih skupin. Vsi ti plani se združijo, uskladijo in rezultat te aktivnosti je

podroben načrt projekta. Pri izdelavi tega plana sodelujejo tako vodja tima, lastnik funkcionalnosti in projektni vodja. Lastnik plana je projektni vodja.

5. Tabela za spremljanje projekta, kjer se redno beležijo statusi posameznih funkcionalnosti na projektu. Lastnik te tabele je projektni vodja.

6. Testni plan, ki opisuje kakšen bo pristop k testiranju na projektu, kakšni so cilji testiranja na projektu in v grobem, kako bodo ti cilji doseženi. Lastnik tega plana je vodja testne ekipe.

7. Testni plan za posamezno funkcionalnost, ki dejansko opisuje, kako se bo katera od novih funkcionalnosti testirala. Ta plan ponavadi napišejo testni inženirji, ki bodo skrbeli za testiranje funkcionalnosti v sklopu projekta.

8. Plan za izdelavo uporabniške dokumentacije, ki opisuje, kateri del uporabniške dokumentacije bo potrebno spremeniti za to, da bodo dodane funkcionalnosti na projektu primerno opisane. Lastnik tega plana je vodja ekipe, ki piše uporabniško dokumentacijo.

9. Plan za upravljanje konfiguracije, cilj katerega je opisati potrebne spremembe na okolju, ki se uporablja za prevajanje in pakiranje produkta. Plan pripravi vodja ekipe za upravljanje konfiguracije.

10. Matrika možnih kritičnih situacij na projektu, ki popisuje te situacije, verjetnost, da do te situacije pride, vpliv na projekt, kateri pogoj mora biti izpolnjen, da se začne plan reševanja te situacije in kakšen je ta plan.

2.2.1. Obseg projekta

Obseg projekta definira stranka v prvi fazi projekta, potem pa se le-ta dokončno določi na osnovi pogajanj med stranko in projektnim vodjem projekta. Glede na to, da je eden izmed pomembnih ciljev projekta tudi čas, v katerem je potrebno dokončati projekt, je potrebno določiti realne cilje, ki se lahko v tem času dejansko dosežejo. Določiti obseg projekta je naloga projektnega vodje, ki se mora uskladiti s stranko. Cilj je določiti množico novih funkcionalnosti, ki jih lahko dokončamo v predhodno določenem časovnem okviru.

Zahteve so določene v dokumentu imenovanem FURPS (Functionality Usability Reliability Performance Supportability), lastnik katerega je stranka. V tem

dokumentu so opisane zahteve stranke glede nove funkcionalnosti z vidika uporabnosti, zanesljivosti, performans in omogočanja podpore. Stranka za vsako od funkcionalnosti, ki je opisana, določiti tudi njeno pomembnost, torej kako pomembno je za stranko, da je funkcionalnost del projekta, katerega obseg določamo.

Prav tako določi prioriteto posamezne funkcionalnosti tudi projektni vodja, katerega glavni vidik pa so inženirji, ki jih ima na razpolago in njihova zasedenost. Tim ima določeno strukturo, določeno število inženirjev, ki imajo ekspertize na specifičnih področjih. Seveda se lahko te ekspertize tudi širijo, tako da inženirji spreminjajo področja, vendar v določenih mejah, tako da ne izgubimo preveč znanja z določenega področja. Tako predstavlja projektnemu vodju struktura tima določeno omejitev, ki jo mora upoštevati pri planiranju.

Potem, ko je določen začetni obseg projekta, projektni vodja skupaj z vodji posameznih timov določi lastnike funkcionalnosti, ki pomagajo projektnemu vodji definirati tabelo obsega projekta, kjer se definira za vsako funkcionalnost nivo potrebne spremembe in vpliv na module. Ta tabela je potem glavno orodje projektnemu vodji pri pogajanjih o obsegu projekta. Iz tabele je jasno vidno, kateri od timov so v tem projektu najbolj obremenjeni. Tako se prvi seznam funkcionalnosti, ki ne bodo prišle v produkt, definira že zelo na začetku projekta, še preden se porabi na teh funkcionalnostih dodaten čas v fazi 2 projekta. Na ta način si lahko prihranimo veliko časa potrebnega za specifikacijo ostalih funkcionalnosti, ki bodo ostale na projektu.

Zahteve za nekatere od funkcionalnosti niso zelo jasne na samem začetku projekta, zato je zanje potreben na začetku določen čas, ki je namenjen raziskavi te zahteve. Med to raziskavo se preveri, kakšne so možne alternative za implementacijo in tudi kakšne so grobe ocene potrebnega dela za implementacijo. Na osnovi rezultatov raziskave se projektni vodja skupaj s stranko odloči, katera alternativa je za ta projekt najprimernejša. Lahko se tudi zgodi, da je zahteva prevelika, da bi jo kar na samem začetku vključili v projekt. V takšnem primeru se odločimo, da bomo zahtevo implementirali ločeno od tekočega projekta. V ta namen se štarta nov podprojekt, v sklopu katerega se poskrbi za to zahtevo. Podprojekt ponavadi nima vseh faz, najbolj pogosto ima samo fazo 2 in 3, potem pa, ko je pripravljen za testiranje v sistemski fazi, se ta funkcionalnost vrne nazaj v glavni projekt. Lahko se zgodi, da to ne bo več tisti projekt, znotraj katerega smo začeli delati na podprojektu. Tako ne bomo po nepotrebnem ogrozili projekta, ki ima jasen cilj v smislu časa, zaradi ene

funkcionalnosti, ki mogoče temelji na tehnologiji, ki še ni popolnoma pripravljena za to, da pride na trg.

Drugi del določanja obsega projekta se naredi znotraj faze 2, ko imamo za posamezno funkcionalnost že pripravljene natančne plane. Lahko, da se izkaže, da še vedno ne moremo narediti vseh planiranih funkcionalnosti v zadanem času. Lahko se zgodi, da ugotovimo, da tudi s trenutnim obsegom ne bomo mogli doseči zadanega cilja projekta. V takšnem primeru imamo seveda dve možnosti, malo podaljšati projekt, če je to še možno s strankinega stališča, ugotoviti, če je katera skupina manj zasedena in uporabiti člana te skupine znotraj druge skupine ali pa zmanjšati obseg projekta.

Seveda pa tudi na koncu faze 2 obseg projekta ni popolnoma dokončno določen. Še vedno lahko stranka spremeni zahtevo. V tem primeru imamo zopet dve možnosti, lahko se zgodi, da povečamo obseg projekta in temu ustrezno spremenimo tudi planirane termine posameznih kontrolnih točk ali pa določene nove funkcionalnosti umaknemo s projekta, tako da lahko ostanemo znotraj predhodno določenih terminskih planov.

2.2.2. Razčlenitev in ocena dela

Delo pri pripravi WBS je razdeljeno v več faz, ki so naštete spodaj in podrobneje opisane znotraj naslednjih razdelkov.

1. WBS je narejena za vsako funkcionalnost, ki je del projekta, v tabeli obsega projekta. To je naloga lastnika funkcionalnosti.
2. Za vse aktivnosti in naloge, ki so definirane znotraj WBS, je potrebno določiti tudi oceno potrebnega dela. Lastnik funkcionalnosti naredi oceno dela na osnovi izkušenj, ki jih ima s preteklih projektov.
3. Definirane WBS za vsako posamezno funkcionalnost se zberejo skupaj v projektni plan. Zbiranje poteka v več fazah. Najprej vodje timov zberejo skupaj WBS za funkcionalnosti za katere imajo lastništvo v timu. Timske WBS se zberejo skupaj v projektni plan dela.
4. Naredi se tudi ocena dela s pomočjo tehnike »od zgoraj navzdol«, kjer planiramo na osnovi analogije s prejšnjimi projekti. Obe oceni, ena je

pridobljena s pomočjo tehnike »od zgoraj navzdol« in druga s pomočjo tehnike »od spodaj navzgor«, se primerjata. Poskušajo se poiskati nekonsistentnosti v planih.

5. Narejena je analiza tabele tveganj.

1. Razčlenitev dela in izdelava WBS

Gre za splošno poznano tehniko, kjer na osnovi razčlenitve potrebnega dela za dokončanje funkcionalnosti in na osnovi ocen dela za vsako od potrebnih aktivnosti na koncu pridemo do ocene dela potrebne za posamezno funkcionalnost.

S pomočjo razčlenitve dela in razdelitve dela na posamezne podnaloge, dejansko veliko bolje razumemo celoten obseg in kompleksnost dela. Prve WBS tabele se pripravijo že v okviru faze 1, ko pripravljamo začetne plane, vendar takrat še niso tako natančne, kot v fazi 2, ko že poznamo načrt za posamezno funkcionalnost in tudi njeno eksterno specifikacijo.

2. WBS v fazi 1

Lastnik funkcionalnosti na osnovi zahteve, ki jo dobi, definira potrebne naloge za to, da se funkcionalnost naredi. Obenem je potrebno tudi okvirno oceniti, kako zahtevna je katera od aktivnosti in koliko dela ocenjujemo, da je za to potrebno.

Za vsako funkcionalnost je potrebno odgovoriti na naslednja vprašanja:

- Ali rabimo posebno raziskavo za zahtevano funkcionalnost, ali pa že imamo dovolj znanja, da pripravimo specifikacijo?
- Koliko dela bo potrebnega za pripravo eksterne specifikacije? Na katerih delih produkta bodo potrebne spremembe?
- Kakšen nivo spremembe zahteva ta funkcionalnost? V primeru, ko gre za bolj zahtevne funkcionalnosti, ali je potreben načrt (HLD - High Level Design) za funkcionalnost? Koncept nivoja spremembe funkcionalnosti, je bolj podrobno razložen v 4.1.1. Tako bo tudi jasno, zakaj ni potreben načrt za vse funkcionalnosti.

- Kolikšno je potrebno delo za to, da dokončamo podroben načrt (LLD – Low Level Design)?
- Kolikšno je potrebno delo za implementacijo zahtevane funkcionalnosti? Na katerih platformah je potrebno implementirati funkcionalnost?
- V kolikšni meri so planirani pregledi kode, v kakšni obliki se bodo izvajali?
- Koliko je potrebnega dela za to, da dokončamo tehnične specifikacije? Ali bo primeren trenutni podroben načrt za tehnično specifikacijo?
- Kako zahtevna bo priprava testnega okolja za izvedbo testiranja? Katere platforme bodo del matrike konfiguracij, ki jih bo produkt podpiral? Kolikšno bo potrebno delo za izvedbo integracijskega testiranja, kolikšno bo potrebno delo za izvedbo sistemskega testiranja?
- Kateri del uporabniške dokumentacije bo spremenjen zaradi dodane funkcionalnosti, kolikšno bo potrebno delo za izvedbo teh sprememb?

Seveda je potrebno imeti obsežno znanje o produktu, da se lahko odgovori na vsa ta vprašanja. Zavedati pa se moramo tudi, da je postavljanje takšne ocene dela timsko delo, kjer si člani tima med seboj pomagajo s svojim znanjem in izkušnjami.

3. WBS v fazi 2

Na osnovi eksterne specifikacije in načrta funkcionalnosti, ki sta rezultat faze 2, imamo na voljo zadosti znanja, da lahko na vprašanja, ki smo si jih zastavili že pri postavljanju ocene v okviru faze 1, sedaj dosti bolj natančno odgovorimo. Ob koncu faze 2 pripravimo bolj natančno WBS za zahtevano funkcionalnost. Zopet je za pripravo WBS za novo funkcionalnost odgovoren lastnik funkcionalnosti, ali pa njegov vodja.

Najprej pogledamo v katerih modulih zahteva posamezna funkcionalnost spremembe in naredimo WBS za vsak modul, ki je ključen za razvoj funkcionalnosti. Lastnik funkcionalnosti je ponavadi član ekipe, ki je lastnik modula, znotraj katerega bodo spremembe največje. Tako je tudi WBS najbolj zahtevna za ta modul. Pomoč pri izdelavi WBS za ostale module lastnik funkcionalnosti dobi pri vodjih timov, ki so lastniki ostalih modulov.

Delo, ki je potrebno znotraj razvojih ekip, je razčlenjeno na:

- delo potrebno za načrt (HLD),
- delo potrebno za podroben načrt (LLD),
- delo potrebno za kodiranje,
- delo potrebno za testiranje enote oziroma modula,
- delo potrebno za preglede kode in njihovo implementacijo pregledov kode in
- delo potrebno za priprava in predstavitev nove funkcionalnosti pred vodstveno ekipo projekta in predstavnikom QA, LP in CPE ekipe.

Delo, ki je potrebno znotraj ekipe za testiranje je razčlenjeno na:

- aktivnosti, ki so potrebne znotraj integracijskega testiranja in
- aktivnosti, ki so potrebne znotraj sistemskega testiranja.

Delo, ki je potrebno znotraj ekipe, ki je zadolžena za uporabniško dokumentacijo, je ravno tako razčlenjeno glede na:

- spremembe, ki so potrebne na uporabniškem priročniku,
- spremembe, ki so potrebne za pomoč uporabniku na grafičnem vmesniku,
- spremembe, ki so potrebne na navodilih za uporabniški vmesnik in
- preglede napisane dokumentacije s strani ostalih članov ekipe, kot tudi s strani razvojnih inženirjev

Iz naštetega je razvidno, da je določitev potrebnega dela zahtevna operacija, za katero so odgovorni izkušeni inženirji, poleg tega pa rabijo pri tem tudi pomoč ostalih članov tima. Rezultat tega procesa je ocena dela potrebnega za določeno funkcionalnost. Potem, ko upoštevamo odvisnosti med posameznimi aktivnostmi, in trajanje posamezne aktivnosti lahko definiramo tudi terminski plan, kdaj bo funkcionalnost pripravljena za določene preglede in kdaj lahko planiramo kontrolne točke, za posamezno funkcionalnost.

Posamezne funkcionalnosti se planirajo neodvisno ena od druge. Vsaka funkcionalnost ima določene svoje kontrolne točke. Ko je ena od funkcionalnosti pripravljena s strani razvojne ekipe gre v integracijsko testiranje neodvisno od drugih funkcionalnosti. Seveda ta princip deluje le, če funkcionalnosti niso medsebojno odvisne.

Vse zbrane WBS za nove funkcionalnost se združijo v plan projekta. Vsota posameznih ocen dela določi oceno dela celotnega projekta. Tej oceni dodamo še ocene dela, ki se planirajo na nivoju projekta in ne na nivoju posamezne funkcionalnosti. Med tovrstno delo sodi:

- delo na upravljanju konfiguracije,
- delo na vodenju projekta,
- delo potrebno za regresijsko testiranje produkta,
- delo potrebno za verifikacijo rešenih problemov na projektu in
- delo, ki se porabi za reševanje problemov prejšnjih verzij produkta. Gre za probleme, ki niso kritični in jih ekipa za vzdrževanje ni popravila v sklopu svojega dela. Končne stranke pa vseeno želijo, da se problem odpravi.

4. Ocena dela za posamezno aktivnost

Ocena dela, ki je potrebno za posamezno aktivnost, je zelo pomemben korak pri postavljanju ocene dela za projekt. Na osnovi razčlenitve dela, ki je potrebno za implementacijo določene funkcionalnosti, je potrebno za vsako od aktivnosti določiti tudi potrebno delo. Vsak tim pripravi oceno dela za aktivnosti, katerih lastniki so. Oceno potrebnega dela določi inženir, ki ima zadosti izkušenj, da lahko na podlagi preteklega dela določi oceno dela.

V primeru, da aktivnost še vedno ni zadosti dobro razčlenjena, je smiselno iti še en korak naprej v razčlenitvi. Manjše aktivnosti lažje razumemo in jim lažje postavimo oceno dela. Ponavadi se ustavimo v procesu delitve takrat, ko je ocena aktivnosti nekje med 1 in 3 dnevi dela. V tem primeru je samemu inženirju lažje razumeti in opravičiti oceno, hkrati pa je ta razlaga tudi na voljo stranki, ki tudi želi razumeti, zakaj neka nova funkcionalnost toliko »stane«.

V trenutku, ko postavljamo oceno, je zelo pomembno, da vemo kateri od inženirjev bo delal na posamezni nalogi, saj se vsi zavedamo, da ne delajo vsi enako hitro. Zavedati pa se tudi moramo, v okviru katerih predpostavk smo naredili oceno. Najbolje je, da se te predpostavke zapišejo in da se definirajo tudi možne kritične situacije v primeru, da se katera od predpostavk ne uresniči. Kot sem že omenila, se takrat, ko se postavi ocena dela za posamezno aktivnost določi tudi inženir, ki bo delal na tej aktivnosti. To je potrebno tudi za to, da se določijo možna ozka grla v planu, v primeru, da smo enega od inženirjev preveč preobremenili z nalogami.

Kaj ponavadi pomaga ocenjevalcu pri njegovem delu?

- Za modul določimo število funkcij, ki jih bo potrebno dodati. Ocenimo delo potrebno za posamezno funkcijo in določimo vsoto posameznih ocen.
- Določimo število testov, ki jih nameravamo izvesti, za to da se prepričamo, da je modul korektno spremenjen, oziroma narejen. Določimo kompleksnost posameznih testov, določimo na katerih platformah se bodo izvajali. Na koncu ocenimo skupno oceno dela, ki bo potrebna za te teste.
- Za postavitev ocene dela na testiranju, zopet določimo koliko testnih scenarijev bo potrebnih, določimo kako bomo pokrili določeno testiranje po posameznih platformah, določimo kolikšen je potreben čas za postavitev testnega okolja in na koncu postavimo kumulativno oceno celotnega dela.
- V primeru uporabniške dokumentacije določimo, kolikšno število strani dokumentacije bo potrebno spremeniti, kakšno je ocenjeno delo za posamezno stran in kolikšna je končna vsota.

Vse ocene, ki se postavijo, najprej pregledajo vodje tima. Pri pregledu ocene dela je potrebno biti še poseben pozoren na naslednje:

- Pogledati je potrebno, če so vse potrebne aktivnosti planirane.
- Preveriti je potrebno smiselnost ocen. Če je na primer zahtevan čas za implementacijo dela funkcionalnosti 5ED, potem si ne moremo privoščiti še enkrat toliko čas za pregled kode.
- Pregledati je potrebno ob kakšnih predpostavkah je bila ocena postavljena in poskušati oceniti verjetnost teh predpostavk.

- Dodati je potrebno še dodatne kritične situacije, ki so možne, pa jih ocenjevalec, mogoče ni znal ali mogel definirati.
- Pomembno je veliko spraševati. Kakšen je pomen te aktivnosti, koliko funkcij bo potrebno spremeniti ali dodati znotraj te aktivnosti, na koliko platformah bo potrebno izvajati testiranje enote in zakaj? Odgovori na ta vprašanja so pomembni za boljše razumevanje ocene, ki jo mora vodja tima na koncu tudi sprejeti, seveda s potrebnimi popravki in jo naprej zagovarjati, ko se te ocene združijo v skupno oceno cene projekta.

5. Usklajevanje posameznih ocen

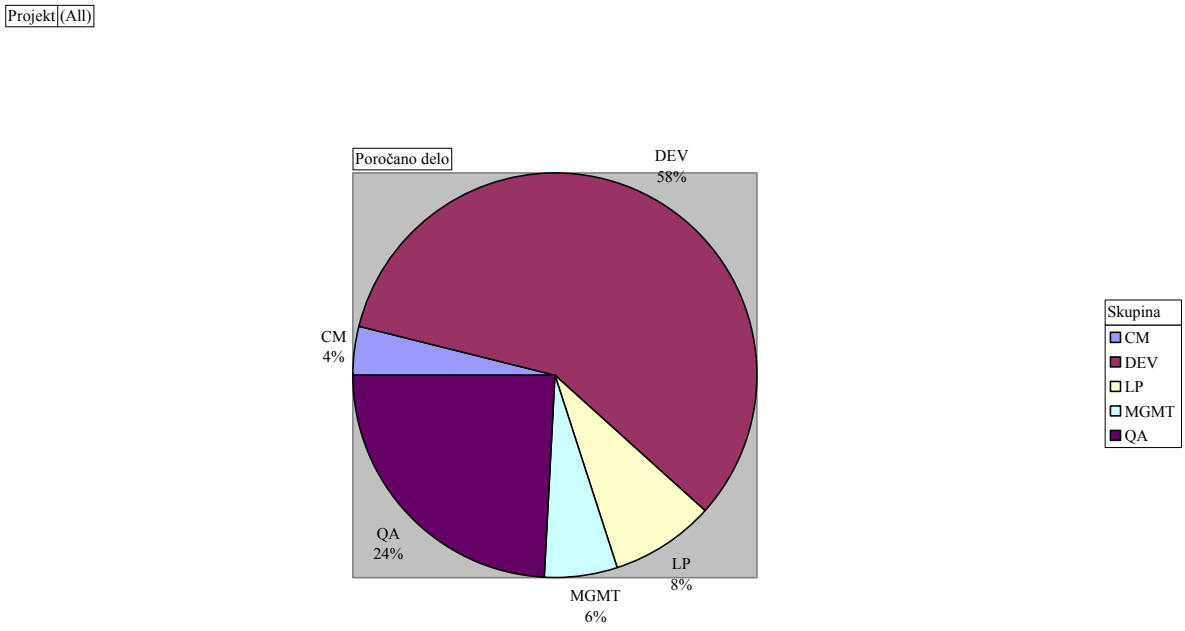
Ko so pripravljene vse ocene za funkcionalnosti vključene v projekt, le-te združimo v skupen projektni plan. Posamezne aktivnostmi v projektnem planu povežemo med seboj glede na njihovo odvisnost in tako pridemo do terminskega plana. Še enkrat se preveri, če se kontrolne točke posamezne funkcionalnosti še vedno ujemajo, sedaj ko upoštevamo zahteve vseh funkcionalnosti hkrati.

Ko vse zahteve uskladimo, pogledamo, katera od funkcionalnosti določa kritično pot na projektu. Za to funkcionalnost poskušamo najti možnosti, kako trajanje kritične poti zmanjšati. Včasih je dovolj že, če tej funkcionalnosti zvišamo prioriteto. Drugič to ni zadosti in kritične poti ne moremo skrajšati drugače, kot če dodamo novega inženirja na izvedbo določenih nalog, čeprav to ni vedno tudi pravi pristop. V primeru, da kritična pot na projektu presega prvotno zadane cilje na projektu, se lahko odločimo ali za podaljšanje projekta ali pa za spremembo obsega projekta.

Potem, ko imamo ta plan, lahko nadaljujemo z usklajevanjem in si pomagamo še s planiranjem s pomočjo tehnike »od zgoraj navzdol«. V naslednjih dveh primerih sta opisana dva primera takšnega ocenjevanja. Recimo, da imamo na voljo ekipo 20 testnih inženirjev, ki bo testirala v fazi sistemskega testiranja. Ta faza bo trajala 2 meseca. 30 EM bomo porabili za naloge, kot so testiranje nove funkcionalnosti in nekaj regresijskega testiranja. Ostane nam 10 EM, ki pa jih bomo porabili za verifikacijo rešenih problemov. Iz predhodnih projektov vemo, da naredimo povprečno 2.5 verifikacije na dan, kar nam da okvirni cilj 500 preverjenih problemov v okviru projekta.

Denimo, da imamo na projektu 40 razvojnih inženirjev. 20% jih bo končalo z implementacijo funkcionalnosti 2 meseca prej kot ostali. Torej imamo na voljo 16EM, ki jih lahko uporabimo za to, da opravimo nekatere pomanjkljivosti, ki so jih stranke prijavile na prejšnjih verzijah projektov.

Pomembni so tudi podatki o preteklih projektih, ki so na voljo. Slika 1 prikazuje, kakšna je bila na preteklih projektih porazdelitev dela med skupinami. Glede na to, da so si projekti podobni, lahko preverimo naše ocene s pomočjo preteklih podatkov in ugotovimo, če kje opazimo nepričakovana odstopanja.



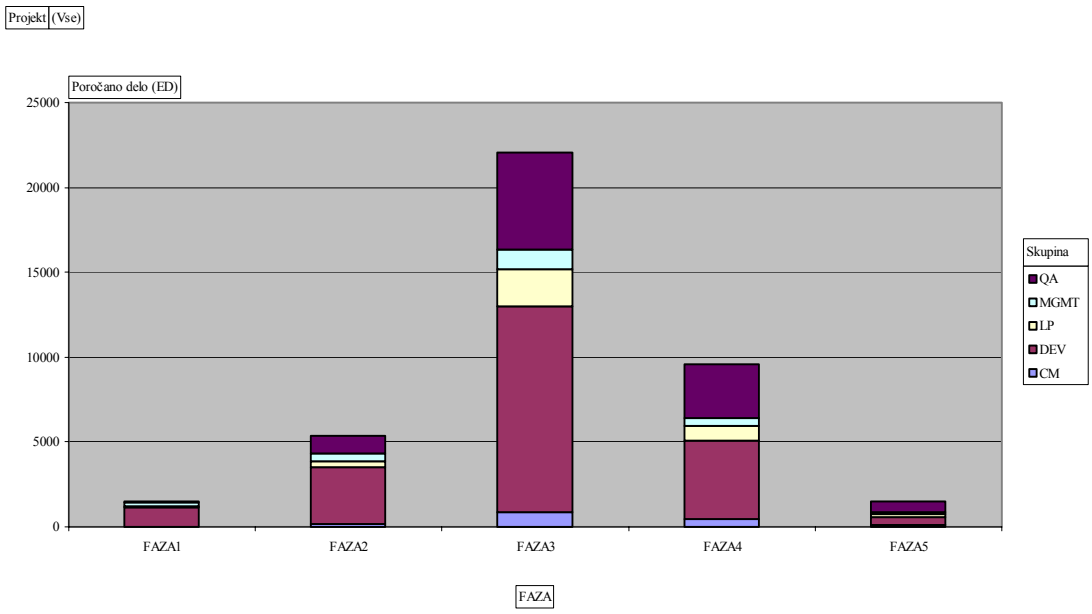
Slika 1: Porazdelitev dela na projektu po skupinah

V primeru, da je na projektu planiranega manj kot 20% dela testne ekipe, se moramo vprašati, če je ta predpostavka dejansko smiselna in res ne rabimo več testiranja. V primeru, kot rabimo več kot 30 odstotkov časa, se moramo vprašati, s čim lahko opravičimo to visoko ceno.

Projektni vodja si naredi plan, ki določa, koliko inženirjev je planiranih na projektu v posamezni fazi. To je pomembno, ker so določeni projekti paralelni in moramo vedeti, kdaj je inženir planiran, da dela na projektu. Da je porazdelitev dela na

projektu po fazah različna lepo kaže tudi Slika 2, ki prikazuje porazdelitev dela med posameznimi skupinami po fazah.

V sklopu prve faze delajo na projektu predvsem projektni vodja in vodje timov, ter lastniki funkcionalnosti. V fazi 2 se vključijo tudi testni inženirji in inženirji, ki delajo na dokumentaciji, ter vodja upravljanja konfiguracije. Vsi delajo pretežno na planih. Faza 3 že obsega sodelovanje vseh ekip, največji delež dela opravijo razvojni in testni inženirji. V fazi 4 je na projektu že enako število testnih inženirjev kot razvojnih inženirjev. V fazi 5 je potrebno le še nekaj dela testnih inženirjev, projektnega vodje in vodje upravljanja konfiguracije.



Slika 2: Delo na projektu po fazah

2.2.3. Terminski plan projekta

Terminski plan projekta lahko določimo, ko razumemo naslednja dejstva:

- približen čas, kdaj želimo produkt dati na trg,
- število inženirjev, ki nam je na voljo in njihova znanja,

- obseg projekta in
- jasna definicija vseh zahtev na projektu.

Prvi terminski plan določimo že znotraj faze 1, kjer definiramo termin za naslednje kontrolne točke:

- konec faze 1, takrat moramo razumeti vse zahteve, definiran mora biti obseg projekta in pripravljen prvi terminski plan.
- konec faze 2, takrat moramo imeti pripravljeno eksterno specifikacijo na projektu, definirani morajo biti potrebni načrti za nove funkcionalnosti in imeti moramo potrjen terminski plan.
- Konec faze 3, takrat sta končani tako implementacija funkcionalnosti, kot tudi integracijski test. Ravno tako je končan del sistemskega testa in pripravljen je produkt beta kvalitete.
- Konec faze 4, takrat je končan prevzem produkta.
- Konec faze 5, takrat je projekt končan in zaprt.

Natančen terminski plan z zagotovili projektne ekipe, da ga je sposobna doseči, se sprejme konec faze 2. Znotraj te faze se definirajo tako kontrolne točke med fazami, kot tudi kontrolne točke, ki se izvajajo znotraj posameznih faz, pa niso tako formalno izvedene kot glavne kontrolne točke. Konec faze 2 se definirajo naslednji mejniki na projektu:

- Dokončana implementacija za vsako funkcionalnost. Takrat lastnik funkcionalnosti pripravi predstavitev, kjer pokaže, da je dodana funkcionalnost dejansko skladna s specifikacijo in jo testni vodja prevzame v integracijsko testiranje.
- Začetek sistemskega testiranja (Beta 0), ko na projektu pripravimo prvo interno beto.
- Konec faze 3, ko na projektu pripravimo prvo eksterno beto (Beta1).
- Pregled novega produkta s strani ekipe za vzdrževanje.
- Konec faze 4 znotraj projektne ekipe v Hermes Softlabu, pripravljena je verzija produkta, ki je namenjena za trg.

- Konec faze 4 znotraj projektne ekipe Hewlett Packard, produkt je pripravljen za na trg.
- Konec faze 5, projekt je končan.

2.3. Zaključek

V tem poglavju so bili opisani glavni koraki, ki so potrebni, da pripravimo plan projekta na razvoju produkta Data Protector. Glede na to, da dela na teh projektih veliko ljudi in da je potrebno veliko dela, da se takšen projekt konča, je potrebno kar nekaj dela tudi za to, da lahko sestavimo projektni plan, ki ga lahko potrdimo stranki. Predvsem je potrebno dobro poznavanje procesov, ki so potrebni, da se vse potrebne aktivnosti pri pripravi plana ustrezno uskladijo.

V nadaljevanju bo opisan model COCOMO II in primer uporabe tega modela na Data Protector projektih. Pokazano bo, da ima smisel uporabiti tudi parametričen način ocenjevanja potrebnega dela pri sestavi plana, kot pripomoček projektnemu vodji, ki lahko na ta način dokaj hitro pride do ocene dela na projektu.

3. COCOMO II

COCOMO II je eden izmed bolj znanih parametričnih modelov, ki nam pomaga pri odločitvah o ceni in planu programskih projektov. Prvotni model imenovan COCOMO (COnstructive COst MOdel) je bil razvit in objavljen leta 1981. COCOMO II je njegov naslednik, razvit kot posledica hitrih sprememb v industriji programske opreme. COCOMO II je tako odgovor na:

- različna nezaporedna izvajanja procesov razvoja programskih produktov,
- principe ponovne uporabe, tako kode, kot tudi ponovni inženiring in uporabo oziroma vključitev že izdelanih programskih produktov,
- principe objektnega programiranja,
- možnost generiranja aplikacije,
- iniciative, ki skušajo izboljšati procese razvoja programske opreme (CMM - Capability Maturity Model).

COCOMO II je razvil konzorcij organizacij, ki ga je vodil Dr. Barry Boehm in nekaj študentov z univerze Južne Kalifornije (University of Southern California USC). Prva verzija modela COCOMO II je bila razvita leta 1996. COCOMO 81 je bil razvit

predvsem za podporo linearnega modela razvoja programskih projektov, medtem ko COCOMO II podpira tudi ostale novejše načine vodenja programskih projektov. Nov model vključuje tudi obnovljeno podatkovno bazo s podatki, ki so jih prispevali člani konzorcija. Model COCOMO II je splošno na voljo vsem potencialnim uporabnikom, dobro dokumentiran [9] in tudi v programski obliki.

Osnovni cilji modela COCOMO II so naslednji:

- Omogočiti določitev natančne cene in plana za tekoče oziroma prihajajoče programske projekte. Pri tem nudi podporo različnim načinom projektnega vodenja, kot so linearni, inkrementalni in ciklični.
- Omogočiti tudi prilagajanje modela, kalibriranje podatkov in razširitev modela, tako da se čim bolj prilagodi zahtevam uporabnika. Rekalibracija omogoča, da uporabnik prilagodi COCOMO II zbranim podatkom v podatkovni bazi. Pri tem lahko določimo način vodenja projekta, način definicije dela, organizacijsko prakso, ...
- Omogočiti jasno definicijo vhodnih in izhodnih podatkov modela, tako da je model natančno določen.
- Omogočiti konstruktivno modeliranje, kjer cena in plan omogočata lažje razumevanje narave projekta, ki ga ocenjujemo.
- Omogočiti nadaljnji razvoj modela, ki podpira nove potrebe oziroma načine dela v programski industriji.

COCOMO II podpira tri nivoje ocenjevanja projektov. Prvi nivo (Application Composition) omogoča oceniti potrebno delo za razvoj produktnega prototipa in uporablja objektne točke kot mero velikosti. Drugi nivo omogoča ocenjevanje v fazi zgodnjega načrtovanja projektov (Early Design). Mera velikosti v tej stopnji so funkcionalne točke oziroma tisoči vrstic kode (KSLOC - Thousands of Source Lines of Code). Tretji nivo je post arhitekturni model (Post Architecture), ki omogoča ocenjevanje projekta po tem, ko smo končali začetno načrtovanje. Tretja stopnja je pravzaprav sprememba modela COCOMO 81.

3.1. Enačba ocene nominalnega plana

Tako post-arhitekturni model (Post Architecture), kot model zgodnjega načrtovanja (Early Design) uporabljata isto formulo za oceno količine dela in koledarskega časa potrebnega za dokončanje projekta. Vhodni parametri modela (Enačba 3-1) so: konstanta *A,* velikost projekta *Size*, eksponent *E* in faktorji dela EM_i (Effort Multipliers). Eksponent *E* nam določajo konstanta *B* in faktorji eksponenta SF_j (Scale Factors). Količina nominalnega dela PM_{NS} ocenjena glede na enoto EM (Engineer Month) se oceni z naslednjo formulo:

$$PM_{NS} = A \times Size^{E} \times \prod_{i=1}^{n} EM_i \qquad \textbf{Enačba 3-1}$$

kjer je $E = B + 0.01 \times \sum_{j=1}^{5} SF_j$

Enačba nominalnega časa $TDEV_{NS}$ (Enačba 3-2), ki določa čas potreben za dokončanje takšnega projekta, je določena z naslednjimi vhodnimi parametri: konstanto *C*, nominalnim delom PM_{NS} in konstanto *F*. Konstanto *F* določajo vrednost konstante *D,* eksponenta *E* in konstante *B* .

$$TDEV_{NS} = C \times (PM_{NS})^{F} \qquad \textbf{Enačba 3-2}$$

kjer je $F = D + 0.2 \times 0.01 \times \sum_{j=1}^{5} SF_j = D + 0.2 \times (E - B)$

Vrednost *n* je 17 v primeru post-arhitekturnega modela in 6 v primeru modela zgodnjega načrtovanja. Tako imamo v prvem modelu 17 faktorjev dela EM_i in v drugem modelu le 6 teh faktorjev. Vrednosti *A, B,* $EM_1, \ldots, EM_{17}, SF_1, \ldots, SF_5$ so dobljene s kalibracijo dejanskih podatkov 161 preteklih projektov v podatkovni bazi COCOMO II. Vrednosti *C* in *D* v modelu COCOMO II določata model projektnega terminskega plana in sta dobljeni s kalibracijo, glede na dejanske vrednosti trajanja 161 projektov v podatkovni bazi.

Vrednosti *A, B,* $SF_1, \ldots, SF_5$ pri modelu zgodnjega načrtovanja so iste kot v prvem modelu. Faktorji dela $EM_1, \ldots, EM_{16}$ so dobljeni s kombinacijo 16 različnih faktorjev dela.

Konstante *A, B, C* in *D* lahko tudi kalibriramo, kar pomeni, da jih prilagodimo lokalnemu okolju in s tem izboljšamo ocene. Vrednosti konstant v samem modelu pa so naslednje:

- $A = 2.94$
- $B = 0.91$
- $C = 3.67$
- $D = 0.28$

Kvocient nominalnega dela in nominalnega časa, potrebnega za dokončanje projekta, nam oceni povprečno število oseb potrebnih za delo na projektu.

Kot vidimo na delo potrebno na projektu vplivajo predvsem naslednje komponente:

- Velikost projekta; podrobnosti, kako ocenimo samo velikost projekta, so opisane v razdelku 3.2.
- Faktorji eksponenta *E*; podroben opis le-teh sledi v razdelku 3.3.1.
- Faktorji dela EM_i, ki so opisani v razdelku 3.3.2.

3.2. Določitev velikosti projekta

Ena izmed najpomembnejših ocen je ocena velikosti projekta. Velikost projekta je najbolj pomembna neodvisna spremenljivka modela, tako da je pravilnost ocene modela v veliki meri odvisna prav od tega parametra. Včasih je določitev velikosti projekta pravi podvig. Ocenjevanje števila vrstic kode je oteženo tudi v primeru, ko ne pišemo nove kode, temveč spreminjamo obstoječo kodo, ali pa uporabljamo avtomatske generatorje kode.

COCOMO II uporablja za izračun vedno vrstice nove kode, tako da se spreminjanje obstoječe kode prevede na število vrstic nove kode. Ta prilagoditev upošteva tako potrebne spremembe v načrtu, kodiranju in testiranju. Upošteva tudi razumljivost kode in pa programerjevo predhodno poznavanje kode.

3.2.1. SLOC

Obstajajo različni načini, kako določiti število vrstic kode (SLOC – Source Lines Of Code). Najzanesljivejši so zbrani podatki v bazi projektov. Tako imamo na osnovi teh podatkov informacijo, kako pretvoriti funkcionalne točke (Function Points), ali kakšno drugo oceno, ki jo lahko podamo v zgodnji fazi projekta, v vrstice kode. V primeru, da baze podatkov nimamo na voljo, lahko uporabimo mnenje eksperta, tako da določi najverjetnejšo, najmanjšo in največjo velikost kode.

Velikost kode se izraža v tisočih vrsticah kode (KSLOC). Ponavadi pri tem ne upoštevamo modulov, ki jih ne dostavimo uporabniku, kot so na primer moduli za testiranje. Vendar pa v primeru, ko jih razvijamo enako natančno kot končno kodo, in ko pri tem uporabljamo iste procese, upoštevamo pri velikosti projekta tudi te module. Cilj je, da merimo količino intelektualnega dela, ki ga moramo vložiti v razvoj projekta.

Definicija števila vrstic kode je zahtevna zaradi konceptualnih razlik med različnimi programskimi jeziki. Težave nastanejo, ko želimo definirati konsistentne načine merjenja za različne programske jezike. V COCOMU II je bil izbran logični stavek kode kot standardna vrstica kode. Pri tem se uporablja definicija SEI (Software Engineering Institute Definition Checklist) za določanje števila vrstic kode. Tako se pri določanju števila vrstic kode upoštevajo programski ukazi, deklaracije, ukazi prevajalniku, ne upoštevajo pa se prazne vrstice, komentarji....

3.2.2. Funkcionalne točke

Ocena funkcionalnih točk temelji na oceni vsebine, oziroma funkcionalnosti projekta in na množici faktorjev projekta. Funkcionalne točke so zelo koristen način cenitve, saj jih lahko določimo že v zgodnji fazi projekta. Tako ocenimo podatke, ki jih procesiramo, eksterne podatke, kontrolne podatke in izhodne podatke. Določimo pet uporabniških funkcionalnih tipov:

- Eksterni vhodni podatki – upošteva vse uporabniške podatke ali uporabniške kontrolne vhode, ki vstopajo v sistem, ki ga merimo.
- Eksterni izhodni podatki – upošteva vse uporabniške podatke ali kontrolne izhode, ki izhajajo iz sistema, ki ga merimo.

- Interna logična datoteka – upošteva vse logične datoteke, ki se generirajo ali uporabljajo v sistemu, ki ga merimo.
- Eksterna vmesna datoteka – upošteva datoteke, ki se uporabljajo kot vmesniki med sistemom, ki ga merimo in drugim sistemom.
- Eksterno povpraševanje – upošteva vse vhodno-izhodne kombinacije, kjer vhodni podatki povzročijo takojšnje generiranje izhodnih podatkov.

Vsakemu primeru, ki sodi v enega od zgoraj naštetih funkcionalnih tipov se določi nivo kompleksnosti. Vsakemu kompleksnemu nivoju se določi seznam tež, ki nam skupaj s številom funkcionalnih točk posameznega tipa določa velikost projekta. Klasična analiza funkcionalnih točk zahteva tudi oceno vpliva 14 lastnosti programske opreme, ki pa je v primeru modela COCOMO II prepovedana, ker se ta vpliv doda v model preko faktorjev dela. Tako štejemo v primeru modela COCOMO II neprilagojene funkcionalne točke UFP (Unadjusted Function Points), ki se uporabljajo kot metrika velikosti. Določanje UPF v COCOMU II sledi naslednji proceduri:

- Določi funkcionalni tip, glede na opis v zgornjem seznamu. To nalogo mora opraviti vodilni inženir na osnovi zahtev projekta in obstoječih dokumentov.
- Vsako funkcijsko točko je potrebno opisati glede na kompleksnost, kot nekompleksno, povprečno kompleksno ali zelo kompleksno, odvisno od tipov podatkovnih elementov, ki jih vsebuje, in od števila datotečnih tipov, ki jih naslavlja.
- Kompleksne nivoje pretvori v teže, ki določajo potrebno delo za implementacijo.
- Seštej vse teže, tako da dobiš eno število UFP.

3.2.3. Relacija med UFP in SLOC

Relacijo med funkcionalnimi točkami in SLOC določimo glede na programski jezik, ki ga uporabljamo. Pri tem uporabimo tabele, ki določajo razmerje med obema

merama. Za programski jezik C je to razmerje 128 vrstic za en UPF, za Java 53, za strojni jezik 320.

3.2.4. Seštevanje nove, pridobljene in ponovno uporabljene kode

Koda, ki je pridobljena iz drugega vira in je uporabljena za razvoj produkta, ravno tako prispeva k velikosti projekta. Obstoječo kodo, ki jo obravnavamo po principu črne skrinjice, imenujemo ponovno uporabljena koda. Kodo, ki jo vklopimo v produkt po principu bele skrinjice, pa imenujemo prilagojena koda. Tako prilagojena, kot ponovno uporabljena koda, se upoštevata pri oceni velikosti projekta. Enota za mero te kode se imenuje ekvivalent vrstic kode (ESLOC - Equivalent Source Lines of Code).

3.2.5. Ponovna uporaba kode

Analize cene ponovne uporabe kode, so pokazale, da je le-ta nelinearna glede na količino ponovno uporabljene kode. Cena ponovne uporabe kode se ponavadi ne začne pri ničli, temveč pri 5% zaradi samega dostopa do kode, izbire in začetnega prilagajanja kode. Izkaže se tudi, da je cena majhnih sprememb v ponovno uporabljeni kodi zelo velika. V tem primeru moramo najprej razumeti program, ki ga spreminjamo in pa njegove vmesnike.

Izkaže se tudi, da je potrebno 47% dela pri vzdrževanju programa porabiti za samo razumevanje modula, ki ga uporabljamo. Takoj, ko želimo nek modul razumeti na principu bele, ne črne skrinjice, moramo pri tem porabiti nekaj časa za to, da ta modul razumemo, hkrati pa tudi module, ki so odvisni od modula, ki ga spreminjamo, in njihove medsebojne vmesnike.

Ceno razumevanja kode in vmesnikov med moduli pa je lahko nižja v primeru, ko je program dobro strukturiran. Modularnost in hierarhična struktura kode lahko bistveno zmanjšata delo, potrebno za ponovno uporabo modula.

3.2.6. Model v primeru ponovne uporabe kode

V primeru ponovne uporabe kode COCOMO II uporablja nelinearni model. Ekvivalent vrstic kode $KSLOC_{EQ}$ se izračuna na osnovi števila vrstic kode, ki so bile prilagojene $KSLOC_{AD}$ in prilagoditvenega faktorja *AAF*. *AAF* določajo trije faktorji spremembe (Enačba 3-4):

- *DM* (Percentage of Design Modified) – sprememba v načrtu izražena v odstotkih,
- *CM* (Percentage of Code Modified) – sprememba v kodi izražena v odstotkih in
- *IM* (Percentage of Integration Modified)– odstotek potrebnega dela za integracijo prilagojene, oziroma ponovno uporabljene kode v nov produkt, in testiranje produkta relativno na integracijo in testiranje, ki so potrebni pri produktu brez sprememb.

$$KSLOC_{EQ} = KSLOC_{AD} \times \left(1 - \frac{AT}{100}\right) \times AAM \qquad \textbf{Enačba 3-3}$$

$$AAF = (0.4 \times DM) + (0.3 \times CM) + (0.3 \times IM) \qquad \textbf{Enačba 3-4}$$

$$AAM = \begin{cases} \dfrac{[AA + AAF(1 + (0.02 \times SU \times UNFM))]}{100}, & AAF \le 50 \\ \dfrac{[AA + AAF + (SU \times UNFM)]}{100}, & AAF > 50 \end{cases} \qquad \textbf{Enačba 3-5}$$

Ekvivalent vrstic kode $KSLOC_{EQ}$ in število vrstic prilagojene kode $KSLOC_{AD}$ povezuje nelinearni faktor (Enačba 3-3). Faktor določajo naslednji parametri:

- *SU* (Percentage of reuse effort due to software understanding) – razumljivost programske opreme, ki je odvisna od same strukture programa, jasnosti in dokumentiranosti. V primeru, da delamo s programom, ki ima jasno strukturo, namen in je dokumentiran, je vrednost faktorja 10 odstotkov, v nasprotnem primeru pa faktor doseže 50 odstotkov.
- *AA* (Percentage of reuse effort due to assessment and assimilation)– stopnja ocene in asimilacije, ki določa potrebno delo za oceno, ali je modul,

primeren za uporabo in vključitev v naš produkt; in pa tudi za integracijo opisa uporabljenega produkta v opis našega produkta. Izražen je v odstotkih. Vrednosti parametra AA segajo od 0 do 8 odstotkov. Najvišji odstotek doseže v primeru, ko je na začetku potrebnega veliko testiranja in evaluacije produkta modulov in dokumentacije.

- *UNFM* (Programmer unfamiliarity with the software)– programerjevo nepoznavanje modula, je faktor, ki določa v kolikšni meri programer pozna kodo, ki bo ponovno uporabljena. Faktor ima vrednost 1, če programer kode sploh ne pozna in 0 v primeru popolnega poznavanja kode.

Ekvivalent vrstic kode se izračuna kot zmnožek med številom vrstic prilagojene kode in faktorjem *AAM* (Adaption Adjustment Modifier), ki določa, kako zahtevna bo ponovna uporaba kode.

Del enačbe $\left(1-\frac{AT}{100}\right)$ se uporabi v primeru, ko imamo opravka z avtomatsko prevedeno kodo, in ga lahko zanemarimo v nasprotnem primeru, ker *AT* postane 0.

3.2.7. Evolucija zahtev (REVL)

Ta faktor omogoča prilagoditi efektivno velikost projekta glede na evolucijo zahtev produkta. Gre za odstotek kode, ki je odstranjen iz produkta. Tako ima projekt velikosti 100 000 vrstic, iz katerega je bilo odstranjenih 20 000 vrstic faktor REVL enak 20. Faktor tako prilagodi velikost produkta v oceni na 120 000 vrstic.

$$SIZE = \left(1+\frac{REVL}{100}\right) \times SIZE_{D,} \qquad \textbf{Enačba 3-6}$$

kjer je $SIZE_D$ velikost projekta na koncu.

3.2.8. Vzdrževanje produkta

V ceno vzdrževanja produkta štejemo tako dodajanje nove funkcionalnosti, kot tudi odpravo napak na stari kodi. V vzdrževanje produkta ne štejemo večjih sprememb na produktu, torej spremembe, kjer spremenimo več kot polovico kode. Velikost vzdrževanja $SIZE_M$ izračunamo z naslednjo enačbo:

$$SIZE_M = [(BaseCodeSize) \times MCF] \times MAF$$ **Enačba 3-7**

Velikost spremenjene kode izračunamo kot produkt velikosti osnovne kode in dveh faktorjev:

- *MCF* (Maintenance Change Factor)– Faktor spremembe na vzdrževanju , izračunamo kot kvocient vsote dodane in spremenjene začetne kode in celotne kode.
- *MAF* (Maintenance Adjustment Factor) - Faktor prilagoditve na vzdrževanju nam omogoča prilagoditi velikost kode glede na faktorje *SU* – razumljivost programske opreme in *UNFM* – programerjevo nepoznavanje produkta.

$$MAF = 1 + \left(\frac{SU}{100} \times UNFM \right)$$ **Enačba 3-8**

3.3. Ocena potrebnega dela

V COCOMU II je potrebno delo izraženo v enoti oseba-mesec *PM* vendar v nalogi uporabljam enoto inženir-mesec *EM*. Inženir-mesec je enota za delo, ki ga ena oseba opravi v enem mesecu pri delu na projektu. Količina dela, ki ga oseba opravi na projektu, je spremenljiva, kot enotna začetna vrednost pa je določeno 152 ur dela na osebo na mesec.

Model za oceno dela je enak za oba modela, tako za model z zgodnjim načrtovanjem, kot tudi za post-arhitekturni model. Določa potrebno delo med dvema mejnikoma v projektnem razvoju programske opreme, od trenutka, ko imamo specificirane zahteve, do trenutka, ko je produkt pripravljen za prevzem. Določajo ga velikost projekta *Size*; konstanta *A*; eksponent *E*; in seznam faktorjev dela EM_i (Enačba 3-1). Število faktorjev dela je odvisno od modela. Konstanta *A* se imenuje konstantna produktivnosti in je določena s pomočjo kalibracije podatkov v projektni bazi in odraža globalno produktivnost. Če model kalibriramo z lokalnimi podatki, ima konstanta *A* vrednost lokalne produktivnosti, kar izboljša natančnost modela. Velikost je določena v *KSLOC*, lahko pa je tudi določena s pomočjo funkcionalnih točk, ki jih potem prevedemo na faktor cene.

Dejavniki, ki vplivajo na ceno – faktorji dela, se uporabljajo za opis karakteristik razvoja programskega projekta, ki vplivajo na potrebno delo na določenem projektu. Vsak izmed dejavnikov ima kvalitativen opis lestvice, ki določa njegov vpliv na potrebno delo v razvoju. Lestvica ima vrednosti od »zelo nizek vpliv«, do »zelo visok vpliv«. Vsaka kvalitativna vrednost na lestvici ima določeno pripadajoče število, ki ga imenujemo faktor cene. Tako prevedemo kvalitativno oceno v kvantitativno, ki jo uporabimo v modelu. Če je faktor cene večji od 1, potem poveča potrebno delo na projektu in obratno faktor cene manjši od 1 zmanjša ceno na projektu.

Ocena faktorja dela je odvisna od skrbnega premisleka, ki vsakega od dejavnikov cene neodvisno opiše in mu določi njegov vpliv na celoten projekt. Izkaže pa se, da je najbolj pomemben dejavnik cene prav velikost projekta, tako da je le-ta dodatno otežena z eksponentnim faktorjem *E*. Ta eksponent je agregacija petih faktorjev lestvice (SF_j), ki jih bomo natančneje razložili v nadaljevanju.

Kar ni jasno iz same enačbe je, da eksponent *E* vpliva na projekt kot celoto in ga zato tudi določamo na projektnem nivoju. Ravno tako tudi zahtevan razvojni plan (*SCED* - Required Development Schedule) določamo samo na projektnem nivoju. Vsi drugi multiplikativni faktorji, kot tudi sama velikost pa se lahko določajo na ravni modulov.

3.3.1. Faktorji eksponenta E

Eksponent *E* v enačbi, ki določa potrebno delo za dokončanje projekta, je agregacija petih faktorjev, kot ga določa Enačba 3-1. Faktor *B* je konstanta, ki ima v modelu COCOMO II vrednost 0.91, lahko pa jo za posamezen produkt tudi kalibriramo. Eksponent *E* določa gospodarnost projekta. V primeru, ko je *E* manjši kot ena, se produktivnost na projektu poveča, če se tudi velikost projekta poveča. Vendar pa to le stežka dosežemo.

V primeru, ko je eksponent *E* enak 1, se za projekt uporablja linearni model. Ponavadi ga uporabljamo za majhne projekte.

Eksponent *E* postane večji od 1 ponavadi zaradi dveh komponent – rast potrebne komunikacije med osebami v ekipi, rast dodatnega dela pri integriranju velikega sistema. Veliki projekti imajo večji projektni tim in zato je potreba po komunikaciji med člani projektnega tima temu ustrezno večja. Prav tako z rastjo produkta narašča

tudi potreba po rasti integracije, ki je potrebna, da vse module na produktu ustrezno preverimo, preden trdimo, da produkt deluje.

$$E = B + 0.01 \times \sum_{j=1}^{5} SF_j$$ **Enačba 3-9**

Vsak faktor, ki ga določamo za sam izračun faktorja eksponenta *E*, določimo iz definirane tabele za model COCOMO II. V faktor E sodijo lastnosti, za katere menimo, da lahko bistveno vplivajo na samo ekonomičnost produkta. Takšen faktor eksponentno vpliva na potrebno delo na projektu, obenem pa tudi na njegovo produktivnost. Vsaka izmed lastnosti ima 6 stopenjsko lestvico, ki jo opisuje. Lastnost lahko na projektu opišemo z naslednjimi opisnimi ocenami od: zelo nizka, nizka, povprečna, do visoka, zelo visoka in izredno visoka. Vsaki opisni oceni, pa je v tabeli pripisana tudi vrednost faktorja lestvice. Vsakega od faktorjev opišemo za dani projekt. Tem opisnim vrednostim poiščemo pripadajoče numerične vrednosti faktorjev v tabeli.

Lastnosti projekta, ki vplivajo na oceno eksponenta *E* v enačbi so naslednje:

- Poznavanje predhodnih projektov (*PREC* - Precedentedness),
- Prilagodljivost razvoja (*FLEX* – Development Flexibility),
- Arhitektura in reševanje kritičnih situacij (*RESL* – Architecture/Risk Resolution),
- Skladnost znotraj ekipe (*TEAM* – Team Cohesion) in
- Zrelost procesov (*PMAT* – Process Maturity).

Na prva dva faktorja nimamo vpliva in jih lahko samo ocenimo, medtem ko se naslednji trije faktorji lahko kontrolirajo in jih lahko z načrtnimi spremembami na produktu izboljšamo.

Vzemimo primer produkta, ki je v vseh lastnostih ocenjen z oceno izredno visoko, ki ji po definiciji v tabeli pripada vrednost 0. V tem primeru je vrednost koeficienta *E* enaka 0.91, torej konstanti *B*. Denimo, da je ocena velikosti projekta enaka 100 *KSLOC*. V tem primeru izračunamo, da je potrebno delo na projektu enako 194 *EM*. V primeru, da bi vse vrednosti ocenili z zelo nizko, je izračunana konstanta *E* enaka 1.226, kar nam da izračunamo delo na projektu 832 *EM*. V primeru, ko bi lahko

uporabili linearen model, bi izračunali potrebno delo 294 *EM*. Ta primer nam pokaže, da ti faktorji lahko pomembno vplivajo na sam potek projekta. Sicer na prva dva faktorja nimamo neposrednega vpliva, lahko pa z naslednjimi tremi bistveno vplivamo na večjo gospodarnost projekta.

V nadaljevanju si bomo bolj podrobno ogledali kriterije, ki opisujejo faktorje.

1. Poznavanje predhodnih projektov (PREC)

Pri določanju faktorja PREC ocenimo:

- kako organizacija razume cilje projekta,
- kakšne so izkušnje pri delu s podobnimi projekti,
- če se skupaj z razvojem programske opreme razvija tudi strojna oprema,
- če se razvijajo novi inovativni algoritmi.

2. Prilagodljivost razvoja (FLEX)

Pri določanju faktorja FLEX ocenimo:

- potrebo po skladnosti programske opreme s predhodno definiranimi zahtevami,
- potrebo po skladnosti programske opreme z eksterno specifikacijo,
- če sta zgornji dve zahtevi povezani tudi z zahtevo, da se projekt hitro konča.

3. Arhitektura in reševanje kritičnih situacij (RESL)

Pri določanju faktorja RESL ocenimo:

- ali obstaja jasen plan ugotavljanja in reševanja kritičnih situacij,
- ali se plan dela, proračun in interni mejniki v razvoju programske opreme skladajo s planom reševanja kritičnih situacij,

- koliko časa, ki ga bomo porabili za sam razvoj, se rezervira za definicijo arhitekture projekta,
- koliko izkušenih arhitektov programske opreme imamo na voljo,
- katera orodja so nam na voljo za reševanje kritičnih situacij, razvoj in verifikacijo arhitekture produkta,
- stopnjo negotovosti za ključne dele arhitekture projekta – uporabniški vmesnik, strojna oprema, tehnologija, zmogljivost,
- število in stopnjo zahtevnosti kritičnih situacij.

4. Skladnost projektne skupine (TEAM)

Pri določanju faktorja TEAM ocenimo:

- konsistentnost skupin, ki se pojavljajo pri razvoju programske opreme v vlogah naročnikov, razvijalcev, testerjev, vzdrževalcev, ...
- pripravljenost prilagajanja med skupinami,
- izkušnje skupin pri skupnem delu,
- druženje skupin, ki omogoči skupno vizijo in večjo sinergijo ekipe.

5. Zrelost procesov (PMAT)

Procedura za določitev zrelosti procesa je skladna s CMM, ki ga je definiral Software Engineering Institute. Upošteva se vrednost faktorja CMM na začetku projekta.

3.3.2. Faktorji dela

V nadaljevanju sledi opis faktorjev dela, ki se uporabljajo v post-arhitekturnem modelu. Faktorje delimo na faktorje produkta, faktorje platforme, faktorje osebja in faktorje projekta. Skupaj model določa 17 faktorjev, ki opisujejo parametre projekta. Vsak od faktorjev je definiran z ocenjevalno lestvico, vsakemu nivoju na lestvici pa

pripada ocena. Nominalna ocena ne vpliva na potrebno delo in ima vrednost 1, nenominalne ocene pa spremenijo potrebno delo, visoka ocena poveča potrebno delo in nizka ocena oceno potrebnega dela zmanjša. Model se uporablja pri razvoju in vzdrževanju projektov aplikacijskih generatorjev, sistemske integracije in infrastrukture.

Faktorje dela ocenjujemo glede na modul, razen faktorja zahtevanega časa razvoja, ki se določi na nivoju celotnega projekta.

1. Faktorji produkta

S faktorji produkta ocenjujemo vpliv lastnosti programske opreme na oceno potrebnega dela. Kompleksen produkt, ki zahteva visoko zanesljivost in dela na veliki podatkovni bazi, zahteva več dela, da ga uspešno razvijemo, kot produkt, ki ima nasprotne lastnosti.

1. **Zahtevana zanesljivost programske opreme (RELY - Required Software Reliability)** – ta določa, kakšne težave nam lahko prinese programska napaka. V primeru, da programska napaka povzroči le manjše neprijetnosti, je ocena zelo nizka, če pa so pri tem ogrožena človeška življenja, je ocena zelo visoka.

2. **Velikost podatkovne baze (DATA – Database Size)** – večja baza zahteva tudi več dela pri generiranju testnih podatkov. Določi se razmerje med velikostjo testne baze (število bitov testne baze) in številom vrstic kode (KSLOC).

3. **Kompleksnost produkta (CPLX – Product Complexity)** – Ta faktor delimo na 5 področij: kontrolne operacije, računske operacije, operacije odvisne od naprav, operacije pri upravljanju s podatki, operacije pri upravljanju z uporabniškimi vmesniki.

4. **Razvijanje produkta za ponovno uporabo (RUSE - Required Reusability)** – Več dela je potrebnega, če želimo produkt pripraviti tudi za to, da ga bomo lahko ponovno enostavno integrirali. Pri takem razvoju moramo napisati bolj natančne specifikacije, obsežnejšo dokumentacijo, izvesti več testov, če ga želimo pripraviti, da se bo lahko še naprej uspešno uporabljal. Faktor RUSE je zato tudi tesno povezana s faktorjema RELY in

DOCU. Modul se lahko ponovno uporabi na več nivojih. V okviru projekta, pri razvoju drugih modulov, v okviru programa, kjer se uporabi pri razvoju večjih projektov za isto organizacijo, v okviru produkte linije, kjer se lahko uporabi v več organizacijah ali pa tudi v okviru več produktnih linij, kot so na primer finance, prodaja, marketing....

5. **Skladnost dokumentacije s trenutnim stanjem projekta (DOCU - Documentation match to life cycle needs)** – pri tem faktorju ocenjujemo, kako se trenutno stanje dokumentacije sklada s trenutnim stanjem projekta. Faktor je zelo nizek v primeru neskladnosti dokumentacije in visok v primeru, ko dokumentacijo ustrezno pokriva razvoj projekta.

2. Faktorji platforme

Platforma opisuje ciljno strojno opremo in programsko opremo, na kateri se bo izvajal naš program.

1. **Omejitev časa izvajanja programa (TIME – Execution Time Constraint)** – izraža kakšno omejitev predstavlja razpoložljiv čas izvajanja programa. Ocena odraža procent razpoložljivega časa, ki ga bomo po pričakovanjih porabili za izvajanje programa. Je nominalna v primeru, ko porabimo samo 50 odstotkov ali manj razpoložljivega časa.

2. **Omejitev glavnega pomnilnika (STOR – Main Storage Constraint)** - Ta ocena izraža omejitev, ki jo predstavlja glavni pomnilnik za izvajanje našega programa. Je nominalna v primeru, ko porabimo samo 50 odstotkov in manj razpoložljivega pomnilnika in zelo visoka, ko je ta procent višji kot 95.

3. **Stabilnost platforme (PVOL - Platform Violability) -** V primeru, ko razvijamo sistem za upravljanje podatkovnih baz, je platforma operacijski sistem in strojna oprema. Vedno gre za sistem, na katerem razvijamo naš produkt.

Faktorji eksponenta		PREC	Poznavanje predhodnih projektov
		FLEX	Prilagodljivost razvoja
		RESL	Arhitektura in reševanje kritičnih situacij
		TEAM	Skladnost znotraj ekipe
		PMAT	Zrelost procesov
Faktorji dela	**Faktorji produkta**	RELY	Zahtevana zanesljivost programske opreme
		DATA	Velikost podatkovne baze
		CPLX	Kompleksnost produkta
		RUSE	Razvijanje produkta za ponovno uporabo
		DOCU	Skladnost dokumentacije s trenutnim stanjem projekta
	Faktorji platforme	TIME	Omejitev časa izvajanja programa
		STOR	Omejitev glavnega pomnilnika
		PVOL	Stabilnost platforme
	Faktorji osebja	ACAP	Sposobnost analitika
		PCAP	Sposobnost programerja
		PCON	Stalnost osebja
		APEX	Izkušnje z aplikacijo
		PLEX	Izkušnje s platformo
		LTEX	Izkušnje s programskim jezikom in orodji
	Faktorji projekta	TOOL	Uporaba programskih orodij
		SITE	Razvoj na več oddaljenih lokacijah
		SCED	Zahtevan plan razvoja

Tabela 1: Seznam vseh faktorjev, ki vplivajo na ceno projekta

3. Faktorji osebja

Po sami velikosti produkta, ki ima največji vpliv na oceno potrebnega dela, so izredno pomembni tudi faktorji osebja. Faktorji so namenjeni ocenjevanju skupine in ne ocenjevanju posameznika.

1. **Sposobnost analitika (ACAP - Analyst Capability) -** Analitik je oseba, ki sodeluje pri specifikaciji zahtev, načrtovanju produkta, ... Pri analitiku predvsem cenimo lastnosti kot so: sposobnost analitičnega

razmišljanja, sposobnost postavitve načrta, učinkovitost in temeljitost in sposobnost komuniciranja in sodelovanja. Ne ocenjujemo pa izkušnje analitika, ker je ta lastnost del posebne ocene.

2. **Sposobnost programerja (PCAP - Programmer Capability) –** ocena mora temeljiti na sposobnosti programerjev kot skupine. Ocenjujemo programerske sposobnosti, učinkovitost in temeljitost, sposobnost komuniciranja in sodelovanja. Ne smemo pa ocenjevati izkušenosti, ker se ta upošteva v ostalih faktorjih.

3. **Stalnost osebja (PCON – Personell Continuity) –** ta ocena je odvisna od odstotka zaposlenih na projektu, ki se letno izmenja.

4. **Izkušnje z aplikacijo (APEX – Applications Experience) –** ocenjujemo izkušenost zaposlenih na projektu s tovrstnimi aplikacijami. Kriterij pri oceni je, koliko časa ekipa že dela s podobnimi aplikacijami.

5. **Izkušnje s platformo (PLEX – Platform Experience) –** ocenjujemo izkušenost zaposlenih s platformo, na kateri razvijamo projekt. Kriterij za oceno je, koliko časa ekipa že dela na platformi, ki jo trenutno uporablja.

6. **Izkušnje s programskim jezikom in orodji (LTEX - Language and Tool Experience)** – pri tem ocenjujemo, kako dobro pozna ekipa programski jezik in orodja, ki jih uporablja.

4. Faktorji projekta

Gre za oceno faktorjev projekta, ki vplivajo na delo. Ti so: uporaba modernih programskih orodji, razvoj programske opreme na več lokacijah in zahtevan plan razvoja.

1. **Uporaba programskih orodji (TOOL - Use of Software Tools) –** programska orodja postajajo čedalje bolj učinkovita in zmogljiva. Ocenjujemo zmogljivost programskega orodja, način integracije in dovršenost.

2. **Razvoj na več oddaljenih lokacijah (SITE - Multi-site operation) –** ocenjujemo način dela ekipe, ki dela na več lokacijah. Pri tem je

pomembno, kaj jim pomaga pri komunikaciji, telefon, elektronska pošta, kakšne so povezave med ekipami,....

3. **Zahtevan terminski plan razvoja (SCED - Required Development Schedule) –** Pri tem faktorju gre za oceno, kako pospešujemo oziroma zaviramo hitrost razvoja glede na nominalni razvoj. V primeru pospeševanja razvoja, je potrebnega več dela na samem začetku, da odpravimo tveganja in pa na koncu, ko želimo paralelizirati čim več aktivnosti, kot so dokumentacija, testiranje.... SCED je edini faktor, ki ga ocenjujemo glede na celoten projekt in ne na modul.

3.3.3. Model zgodnjega načrtovanja

Ta model se uporablja v zgodnjih fazah projekta, ko zelo malo vemo o velikosti produkta, ki ga nameravamo razviti, o platformi, kjer bomo razvijali, o izkušenosti ekipe, ki bo sodelovala na projektu. Tudi ta model uporablja KSLOC in prilagojene funkcijske točke za velikost. Faktorji lestvice so enaki, kot pri post-arhitekturnem projektu. Faktorji cene pa se glede na post-arhitekturni model združujejo. Pri tem združimo faktorje projekta:

- RELY, DATA, CPLX, DOCU v faktor zanesljivosti in kompleksni **RCPX (Product Reliability and Complexity)**.
- Faktor razvijanja produkta za ponovno uporabo ostane isti **RUSE**.
- Faktor zahtevnost platforme (**PDIF – Platform Difficulty**) se sestavi iz faktorjev TIME, STOR, PVOL.
- Faktor sposobnost ekipe (**PERS – Personell Capability**) se sestavi iz faktorjev ACAP, PCAP, PCON.
- Faktor izkušenost ekipe (**PREX - Personnel Experience**) se sestavi iz faktorjev APEX, PLEX, LTEX.
- Pomagala (**FCIL - Facilities**) tvorita dva fakorja TOOL, SITE.
- Faktor zahtevan plan razvoja ostane **SCED.**

3.4. Ocena potrebnega dela v primeru razvoja več modulov

V primeru, ko želimo oceniti delo na več modulih ocena potrebnega dela ne more biti preprosto vsota vseh potrebnih modulov, saj bi v tem primeru zanemarili ceno, ki je potrebna pri integraciji posameznega projekta. Tehnika, ki je opisana spodaj se uporablja v primeru, ko smo produkt razčlenili na *n* modulov na enem nivoju.

1. Izračunaj vsoto vseh komponent, da dobiš skupno vsoto.

$$Size_{Aggregate} = \sum_{i=1}^{n} Size_i \qquad \textbf{Enačba 3-10}$$

2. Uporabi parametre, ki se upoštevajo na nivoju projekta na skupni vsoti, tako da dobiš splošno oceno potrebnega dela PM_{Basic}.

$$PM_{Basic} = A \times \left(Size_{Aggregate}\right)^{E} \times SCED \qquad \textbf{Enačba 3-11}$$

3. Določi za vsako komponento potrebno osnovno delo za dokončanje modula, tako da razdeliš osnovno celotno delo na module po relativni velikosti.

$$PM_{Basic(i)} = PM_{Basic} \times \left(\frac{Size_i}{Size_{Aggregate}} \right) \qquad \textbf{Enačba 3-12}$$

4. Uporabi parametre, ki se določajo na nivoju posameznega modula, za vsak modul.

$$PM_i = PM_{Basic(i)} \times \prod_{j=1}^{16} EM_j \qquad \textbf{Enačba 3-13}$$

5. Izračunaj skupno vsoto posameznega dela po modulih.

$$PM_{Aggregate} = \sum_{i=1}^{n} PM_i \qquad \textbf{Enačba 3-14}$$

Oceno potrebnega časa dobimo tako, da ponovimo korake od 2 do 5 brez SCED parametra, ki se je uporabil v drugem koraku. Iz $PM_{Agregate}$ po enačbi 3-2 izračunamo potreben čas.

4. Uporaba modela COCOMO II na projektu Aragon

Klasičen način ocenjevanja cene projekta je bil opisan v poglavju 2. V okviru tega poglavja pa bom opisala, kako sem prilagodila parametričen model COCOMO II, model za ocenjevanje projektov, na razvoju produkta Data Protector. Prvi pogoj, da se lahko lotimo definirati parametrično oceno za produkt je, da so nam na voljo zgodovinski podatki. Glede na to, da se programski produkt razvija že več kot 10 let, je na voljo kar nekaj zgodovinskih podatkov. Res pa je, da na samem začetku ni bil določen nek standard, kako se bodo ti podatki zbirali in je bilo potrebno vložiti kar nekaj dela, da sem sestavila podatke, ki so primerni kot osnova za parametrično oceno. Podatke za vse projekte sem zbirala na enak način, tako da so medsebojno primerljivi.

Za to, da sem lahko model COCOMO II prilagodila ocenjevanju projektov, so bili potrebni naslednji koraki:

1. Prvi pomemben korak je bil določiti velikost projekta. Tipična lastnost Data Protector projektov je, da vedno vzamejo za osnovo dokončano delo na

preteklem projektu. Vsak naslednji projekt podeduje od prejšnjega tako izvorno kodo, sistem za prevajanje in pakiranje produkta, okolje na katerem je nameščen sistem za prevajanje, tehnično dokumentacijo, uporabniško dokumentacijo, testno okolje skupaj z že napisanimi testnimi moduli in scenariji. V okviru novega projekta se podedovano okolje starega projekta prilagodi novim zahtevam projekta. Tako je bila ena od dilem, kako dejansko določiti velikost projekta. Ali so vrstice kode napisane na takšnem projektu enako zahtevne, kot vrstice kode na projektu, ki nima takšnega podedovanega okolja?

2. Drugi korak, ki ga je bilo potrebno narediti, je bil proučiti faktorje eksponenta in faktorje dela, ki jih je potrebno določiti pri uporabi modela COCOMO II za tekoči projekt. Faktorje sem določila za projekt, ki je bil v času nastajanja te naloge aktiven, to je projekt Aragon, znotraj katerega se je razvijala nova verzija programske produkta Data Protector.

3. Tretji izmed pomembnejših korakov je bil preslikava življenjskega cikla projekta, ki ga uporablja COCOMO II, na življenjski cikel projekta, ki se uporablja na projektih v okviru razvoja produkta Data Protector. Ta korak je narejen predvsem za to, da imamo boljši občutek, kakšne so razlike med obema modeloma in da bolj plastično predstavimo potrebo po kalibraciji modela na osnovi lokalnih podatkov.

4. Naslednji zelo pomemben korak pa je bil prilagoditev modela COCOMO II razmeram, ki vladajo pri razvoju projektov, torej kalibracija modela COCOMO II. Samo kalibriran model nam lahko da, na samem koncu, parametrično oceno, ki je dejansko v skladu z načinom dela, ki poteka v okolju, za katerega poskušamo prilagoditi model. Pri tem koraku sem si pomagala s produktom CALICO 7.0 (CALIbrate COcomo).

Kot rezultat vseh naštetih korakov je nastal model, s katerim si lahko pomagamo pri ocenjevanju naslednjih projektov na razvoju projekta Data Protector. Seveda to ne pomeni, da se bomo odpovedali klasičnemu načinu ocenjevanja projekta. Model nam določi velikostni razred projekta, omogoča nam razumeti, kako vplivajo razni faktorji na projekt. Še vedno pa bomo uporabljali tudi način ocenjevanja, ki smo ga uporabljali do sedaj. Primerjava obeh ocen nam omogoča prilagajanje obeh modelov in postavitev končne ocene, ki se uporabi v pogajanjih za ceno projekta.

4.1. Določitev velikosti projekta

Kot sem že omenila v uvodu, gre pri razvoju programskega produkta Data Protector za spreminjanje obstoječega produkta. Vsak nov produkt podeduje rezultate prejšnjih produktov in nadaljuje razvoj programskega produkta. Model COCOMO II pa opisuje razliko med projekti, ki ponovno uporabijo kodo in med tistimi, ki te začetne zasnove nimajo. Vprašanje, ki sem si ga zastavila na samem začetku je, ali lahko ocenimo, da so vrstice kode, ki jih pišemo, ekvivalentne vrsticam novo napisane kode? Kot odgovor na to sem postavila model, ki naredi preslikavo med vrsticami kode, ki so bile na projektu podedovane, v vrstice kode, ki so ekvivalentne na novo napisanim vrsticam kode. Osnova za ta model je COCOMO II model v primeru ponovne uporabe kode opisan v razdelku 3.2.6.

V bistvu se na začetku vsakega novega projekta vzame koda, dokumentacija in vsa obstoječa infrastuktura preteklega projekta kot osnova, na tej bazi pa se potem gradi množica novih funkcionalnosti. Pojem »nova funkcionalnost« je zelo širok pojem, ki dejansko označuje katerokoli spremembo na produktu, ki jo lahko zaznamo v novi verziji produkta. Pri tem pa gre lahko za zelo različne kategorije dela, ki je lahko bolj ali manj kompleksno.

Včasih je nova funkcionalnost samo preprosto prilagajanje obstoječega modula na novo verzijo operacijskega sistema. V drugem primeru gre lahko za dodajanje funkcionalnosti, ki bo podprla novo tehnologijo shranjevanja podatkov. Za to so potrebne že bolj korenite spremembe, ki zahtevajo več sprememb na samem načrtu produkta. V takšnem primeru so spremembe potrebne tudi na več modulih produkta. Nova funkcionalnost pa lahko zahteva tudi spremembo same arhitekturo produkta. V takšnem primeru je potrebno še bolj globalno poseči v samo jedro produkta in spremeniti obstoječe temelje samega produkta. V nadaljevanju si bomo pogledali, kako smo na projektu Aragon uporabili model ponovne uporabe kode, kot mehanizem za določanje velikosti produkta.

4.1.1. Nivo spremembe funkcionalnosti

Še preden smo se odločili za samo merjenje velikosti produkta, smo imeli že izdelan koncept, ki nam je omogočal klasifikacijo novih funkcionalnosti na osnovi zahtev, ki so določale novo verzijo produkta. Ta klasifikacija nam je omogočila, da

smo lažje razumeli samo naravo spremembe, in da smo obenem tudi lažje planirali in dodeljevali naloge, ki jih je bilo potrebno izvesti za to, da so spremembe prišle v sam produkt. Pri tem se je potrebno zavedati, da je tovrstno delo zahtevno, saj je programski produkt Data Protector izredno velik, obsega seznam različnih funkcionalnih modulov, ki so med seboj bolj ali manj odvisni. Za to, da projekt razumemo, je potrebno dobro poznati njegovo strukturo in njegove omejitve. Vsako novo zahtevo na produktu je potrebno dobro razumeti, zato da so v trenutku, ko želimo to spremembo dati v produkt in nadaljnje testiranje, vse spremembe jasno definirane, opisane in dodane. Eden od pomembnih korakov, ki jih je potrebno narediti za to, da funkcionalnost razumemo, je določitev nivoja potrebne spremembe. Klasifikacija pa zelo jasno določa, na katerih modulih se bodo dogajale spremembe in v kakšni meri.

Definiranih je 5 različnih nivojev spremembe:

- Nivo 0 – dodana funkcionalnost ne zahteva nobene spremembe kode. Tipično gre v takem primeru za certifikacijo obstoječe funkcionalnosti na novi verziji programske opreme, s katero je ta funkcionalnost integrirana. Edina dodatna spremembe, ki je še potrebna, je sprememba matrike, ki določa seznam podprtih platform in aplikacij. Ta matrika pa se ne vodi kot del uporabniške dokumentacije.

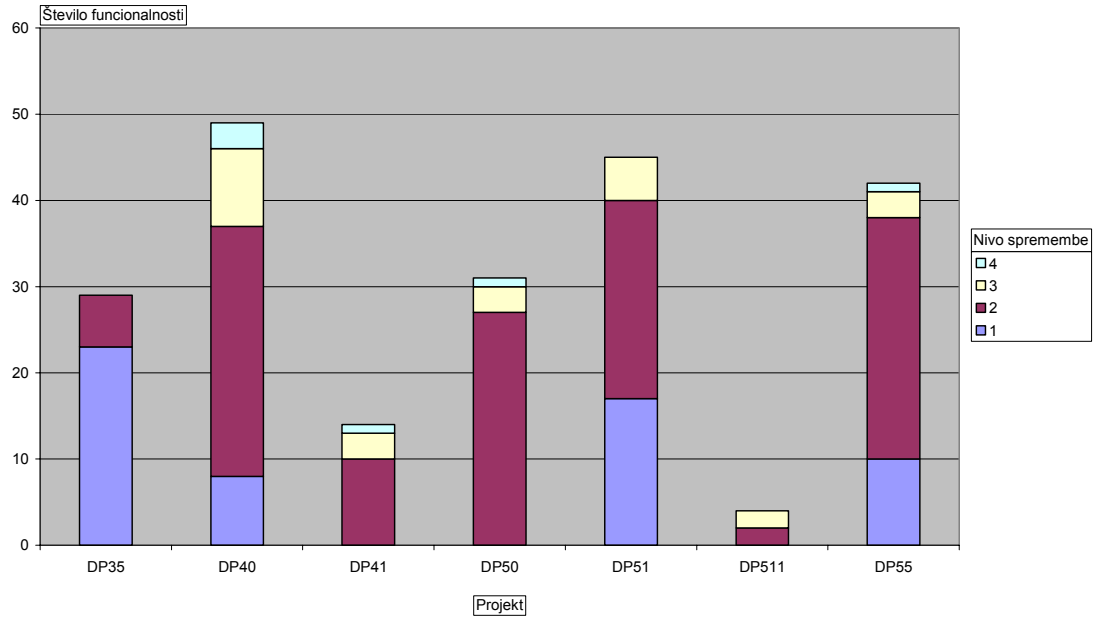
- Nivo 1 – dodana funkcionalnost zahteva samo spremembo nekaj vrstic kode znotraj enega od modulov. Pomembno je, da ne zahteva nobene spremembe uporabniške dokumentacije. Tudi v tem primeru gre lahko za certifikacijo obstoječe funkcionalnosti na novi verziji programske opreme, ali pa za prilagajanje obstoječega modula na novo platformo. Zopet je potrebna sprememba matrike, ki določa seznam podprtih platform in aplikacij.

- Nivo 2 – dodana funkcionalnost zahteva spremembo kode znotraj več različnih modulov. Vse spremembe, ki so potrebne, so še vedno v skladu z obstoječimi koncepti produkta. V tem primeru ponavadi obstoječemu modulu dodamo nekaj nove funkcionalnosti. Spremembe so potrebne tudi v uporabniškem vmesniku, dokumentaciji in ostalih modulih produkta.

- Nivo 3 – dodana funkcionalnost zahteva velike spremembe. Potrebno je definirati nove koncepte, dodati podporo za nove tehnologije. Spremembe so zahtevane znotraj različnih modulov. V takšnem primeru gre lahko tudi za

razvoj novega podmodula. Primer takšne funkcionalnosti je, na primer razvoj nove integracije, ki omogoča varnostno shranjevanje podatkov podatkovne baze s pomočjo programskega produkta Data Protector. V tem primeru bomo razvili tako nov podmodul na modulu integracije, spremembe pa bodo potrebne tudi v drugih modulih, na primer v uporabniškem vmesniku, programih za namestitev in uporabniški dokumentaciji.

- Nivo 4 – funkcionalnost zahteva arhitekturne spremembe. V tem primeru spreminjamo temelje produkta. Kot primer lahko navedemo spremembe, potrebne v podatkovni bazi produkta, ki nam bodo omogočile, da podatki lahko narastejo preko trenutno določenih mej. V takšnem primeru se lahko odločimo le za razširitev obstoječega modela, ki mu odpravimo omejitev ali pa za postavitev novega modela podatkovne baze, za kar je potrebno samo arhitekturo podatkovne baze spremeniti.

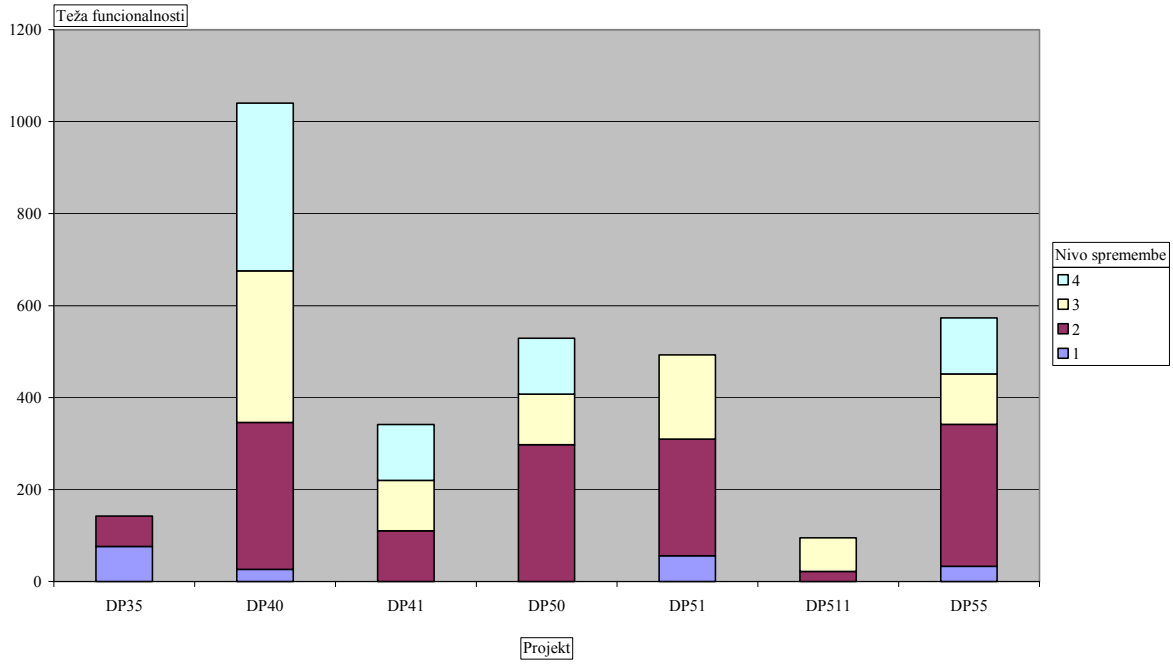


Slika 3: Število dodanih funkcionalnosti glede na nivo spremembe

Tipično se v okviru nove verzije programskega produkta razvije nekaj 10 funkcionalnosti kompleksnosti 0, 1, 2. Število dodanih funkcionalnosti reda 3 in 4 pa je bolj omejeno, ker so v nasprotnem primeru spremembe na produktu prevelike. V novi verziji produkta se lahko doda okrog 5 funkcionalnosti kompleksnosti nivoja 3 in

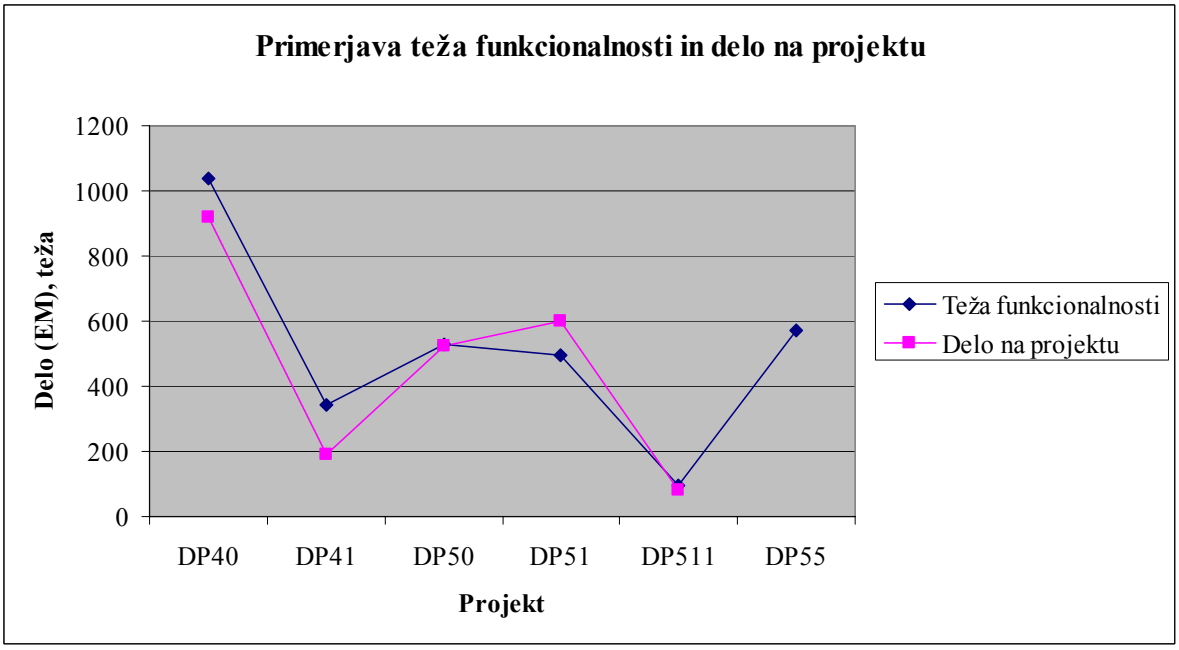
le ena do dve funkcionalnosti kompleksnosti nivoja 4, če sploh. Ponavadi se takšne funkcionalnosti razvijajo kot podprojekti, tako da zmanjšamo tveganje trenutno aktivnega projekta. Slika 3 prikazuje število dodanih funkcionalnosti na projektu za pretekle projekte.

Slika 4 pa nam ocenjuje težo funkcionalnosti, ki smo jo dodali na posameznem projektu. Kot mero za oceno teže projekta glede na dodane funkcionalnosti, sem določila eksponentno funkcijo, ki nam ocenjuje težo dodanih funkcionalnosti na projektih. Zanimivo je, da je ta mera dobro usklajena s samim delom, ki se je vložilo v projekt, kar nam prikazuje Slika 3. Tako nam lahko ta relativno enostaven model, že na samem začetku prikaže kar nekaj informacije o tem, koliko dela bo potrebno vložiti v projekt.



Slika 4: Teža dodanih funkcionalnosti glede na nivo spremembe

Če upoštevamo, da je korelacija med težo dodanih funkcionalnosti in delom vloženim v produkt visoka, potem lahko ocenimo, da se bo cena projekta Aragon gibala nekje med 600 in 700 EM. Glede na trenutno situacijo na projektu, kjer se dodajajo v zelo pozni fazi, konec faze 3, še nove zahteve, pa bi lahko ocenili, da bo lahko ta cena še višja.



Slika 5: Primerjava teža funkcionalnosti in delo na projektu

4.1.2. Določitev vpliva na module

Drugi pomemben vidik pri analizi teže novega produkta je vpliv na module, ki tvorijo programski produkt Data Protector. Pri vsaki dodani funkcionalnosti je potrebno razumeti, znotraj katerih modulov so potrebne spremembe in kako kompleksne so. Za nov projekt se naredi matrika funkcionalnosti in modulov. Programski produkt Data Protector tvorijo naslednji programske moduli:

1. Vrstični vmesnik (CLI – Command Line Interface).
2. Skupna knjižnica (CORE – Core Library).
3. Upravljalec sej (CS – Cell Server).
4. Agent za branje/pisanje podatkov na/z diska (DA – Disk Agent).
5. Podatkovna baza (DB - Database).
6. Agent za obnovitev sistema (DR – Disaster Recovery).
7. Grafični vmesnik (GUI - Graphical User Interface).
8. Inštalacija (INST - Installation).

9. Agent za varnostno shranjevanje podatkovnih baz (INT - Integrations).
10. Agent za branje/pisanje podatkov na/s tračne enote (MA – Media Agent).
11. Agent za varnostno shranjevanje preko konsistentne kopije (ZDB – Zero Downtime Backup)
12. Ostalo (OTH - Other).

Vsaka funkcionalnost ima določenega tudi lastnika te funkcionalnosti. Tipično je to član skupine, ki dela na modulu, na katerem bo potrebnih največ sprememb za novo funkcionalnost. V večini primerov, če gre za funkcionalnost, ki zahteva nivo spremembe 0-2, so spremembe potrebne le v enem od modulov. V primeru bolj kompleksnih funkcionalnosti, ki zahtevajo nivo spremembe 3-4, pa so potrebne spremembe v več modulih, vendar so tipično nosilci glavnih sprememb le eden do dva modula. Tabela 2 spodaj prikazuje, kako so bile zahteve po spremembah porazdeljene po modulih na projektu Data Protector 5.5.

Naslednji pomemben vidik je tudi, v kolikšni meri bo funkcionalnost vplivala na module, ki se uporabljajo za testiranje produkta. V kolikšni meri se bodo lahko uporabili obstoječi testni moduli in se bo testiranje le ponovilo, ter v kolikšni meri bo potrebno napisati nove testne scenarije za testiranje nove funkcionalnosti. Prav tako se določi, v kakšni meri bodo potrebne spremembe na uporabniški dokumentaciji. Kje bo potrebno napisati nova poglavja, mogoče dodati tudi nove priročnike, ali pa gre le za manjše spremembe dokumentacije. Vse te analize so pomembne v naslednjem koraku, ko poskušamo določiti velikost produkta.

ID	CORE	CS	MA	DA	GUI	CLI	INST	INT	ZDB	OTH	Nivo
FDR24934							X				2
PRO15904	X							X			2
FDR16639	X				X						2
GRC24922		X			X						2
FDR15986		X									2
OPR21616		X				X	X				2
HSL00004		X	X	X	X			X	X		3
HSL00001	X		X		X	X	X				4
HSL00003	X		X	X	X	X	X	X	X		2
OPR21089	X		X	X							2
SPT21611	X					X	X				2
USR10718					X						2
FDR20802					X						2
FDR21160							X				2
FDR21634							X				2
SPT23931							X				3
HSL00005								X			2
FDR13933	X				X	X	X	X			3
FDR23458	X							X			2
FDR23848	X				X			X			2
FDR21516										X	1
FDR21650										X	1
USR21436										X	1
FDR14150			X	X							2
FDR20798			X								2
FDR20737			X	X							2
IN25675			X								2
FDR25674			X								2
FDR10702			X		X						2
FDR10696	X		X		X	X					2
USR15976					X						1
USR15975					X						1
USR15263					X						1
MPR26025										X	1
FDR26023										X	1
GRC21612	X		X	X	X	X	X	X	X		2
GRC21644	X		X	X	X	X	X	X	X	X	2
LGL29056	X										1
FDR28011	X				X						2
FDR17228								X			1
FDR17229								X			2
FDR27385			X								2

Tabela 2: Seznam funkcionalnosti in vpliv na module

4.1.3. Določitev DM

Za vsakega od Data Protector modulov je potrebno določiti, kakšna je sprememba načrta, potrebna na trenutnem projektu. Pri tem si pomagamo s tabelo funkcionalnosti, ki določa, katere od funkcionalnosti imajo vpliv na katerega od modulov in kakšen je nivo spremembe, potreben pri določeni funkcionalnosti. Pomembna je tudi informacija o modulih, ki so nosilci spremembe. Odvisno od zgoraj naštetih kriterijev , se za vsakega od modulov določi faktor spremembe načrta DM.

Pri določanju faktorja DM si lahko pomagamo s spodnjo preslikavo. V primeru, da so na modulu dodane le funkcionalnosti 0-1, ne moremo govoriti o potrebni spremembi načrta. Nasprotno pa je sprememba načrta velika v primeru, ko dodajamo v produkt funkcionalnost, ki zahteva spremembo arhitekture projekta. Spodaj je nekaj smernic, ki nam pomagajo pri določanju faktorja spremembe produkta, ki pa jih moramo razumeti zgolj kot priporočila.

- Nivo 0 – DM 0 : v primeru, ko ne potrebujemo niti spremembe kode, zagotovo ne potrebujemo niti spremembe načrta.
- Nivo 1 – DM 0 : v primeru, ko je potrebno spremeniti le nekaj vrstic kode, ne moremo govoriti o zahtevah spremembe načrta.
- Nivo 2 – DM 0-5 : v primeru, ko so potrebne spremembe v več modulih, koncepti pa še vedno ostajajo isti, lahko že opravičimo delno spremembo načrta, vendar gre tu ponavadi za manjše spremembe, ki so bolj interne in niso vidne na samih vmesnikih med moduli.
- Nivo 3 – DM 10-30 : v primeru, ko dodana funkcionalnost zahteva podporo različnih konceptov, je seveda temu potrebno prilagoditi tudi načrt. Tako v tem primeru lahko opravičimo večje spremembe na samem načrtu.
- Nivo 4 – DM 30-100: logična posledica spremembe arhitekture je tudi sprememba načrta. V tem primeru je faktor spremembe načrta največji.

4.1.4. Faktor spremembe kode CM

Za vsako izmed naštetih funkcionalnosti, je potrebno tudi določiti faktor spremembe kode CM. Postopek za določitev faktorja spremembe kode je dokaj

preprost. V tej fazi že razumemo, kateri moduli programskega produkta bodo spremenjeni. Za vsakega od teh modulov imamo podatek o njegovi velikosti, torej o vrsticah kode. Sedaj je potrebno določiti, v kolikšni meri se bodo te obstoječe vrstice kode spreminjale. Zopet si pomagamo z matriko funkcionalnosti in vplivom na module in določimo, v kakšni meri pričakujemo, da se bo število vrstic na modulu spreminjalo. Vsota vseh sprememb, ki se na modulu pričakujejo na koncu, določa faktor spremembe kode na modulu.

Obstaja tudi korelacija med faktorjem spremembe kode in določenim nivojem funkcionalnosti:

- Nivo 0: CM 0; v tem primeru je tudi faktor spremembe kode enak 0.
- Nivo 1: CM 0; v tem primeru je faktor spremembe kode majhen, nekaj vrstic, tako da ga lahko zanemarimo.
- Nivo 2: CM 0-5; v tem primeru je lahko faktor spremembe že nekoliko večji, vendar tipično ne bi smel presegati 5 odstotkov.
- Nivo 3: CM 5-20; v tem primeru je faktor spremembe lahko že bolj visok. Vzemimo primer, ko razvijamo novega integracijskega agenta. Vendar pa je nov integracijski agent le eden od podmodulov na integracijskem modulu.
- Nivo 4: CM 20-50; v tem primeru smo se odločili, da bomo spremenili samo arhitekturo modula. Posledično lahko pričakujemo tudi velike spremembe na kodi. Tudi tu je lahko ocena faktorja IM visoka, seveda v modulu, kjer se bo ta sprememba dejansko implementirala.

4.1.5. Faktor spremembe integracije IM

Pri faktorju spremembe integracije nas zanima, kakšna bo potrebna sprememba, da bomo novo funkcionalnost uspešno integrirali v produkt. V bistvu se vprašamo, v kolikšni meri nameravamo, glede na predhodno verzijo produkta, spremeniti način testiranja, v smislu dodajanja novih testnih modulov, da bomo zagotovili ustrezno kvaliteto.

Mogoče bo koncept še najbolj jasen, če si ga ogledamo na primerih, ker tu zelo težko določimo splošne napotke. Dejansko je dejstvo, da je potrebno v primeru

produkta Data Protector ponoviti visok odstotek testiranja, zato da lahko zagotovimo, da se funkcionalnost primerno integrira v produkt in obenem ne pokvari že obstoječe funkcionalnosti, ki je na voljo že od prejšnjih verzij produkta. Pri tem je potrebno razumeti, da je produkt izredno velik in da vedno ne moremo ponoviti vseh testov, na vseh platformah. Ponavadi se tu odločimo za neke vrste kompromis, saj so integracijske matrike, ki dejansko popisujejo različne možne konfiguracije pri funkcionalnosti, ki jo testiramo, v veliki meri preobsežne. Kot primer si poglejmo parametre, ki vplivajo na testiranje novo dodane funkcionalnosti na integracijskih agentih, ki omogočajo »point in time« shranjevanja podatkov. Seznam je kar dolg:

1. verzija operacijskega sistema, na katerem teče podatkovna baza oziroma aplikacija,
2. vrsta integracije s podatkovno bazo ali aplikacijo,
3. verzija integrirane aplikacije,
4. vrsta »point in time« agenta, odvisno na katerem diskovnem polju deluje,
5. tip načina shranjevanja podatkov, lahko gre za direktno shranjevanje na disk, lahko gre za shranjevanje na trak, lahko za kombinacijo obeh.

V tako kompleksni matriki je nemogoče testirati vse možne kombinacije, ampak se ponavadi odločimo za smiselne nabore kombinacij. Testiranje teh kombinacij naj bi s čim večjo verjetnostjo zagotovilo, da smo dodano funkcionalnost dobro testirali in integrirali v produkt.

Pri določanju faktorja IM si zopet pomagamo z matriko funkcionalnosti in modulov. Pri tem določamo predvsem, kako se bo testiranje modula spremenilo glede na že obstoječe teste, če dodamo v produkt funkcionalnosti, ki so planirane na tekočem projektu.

Če primerjamo faktor spremembe načrta in faktor spremembe kode, lahko ugotovimo, da oba naraščata z nivojem kompleksnosti funkcionalnosti. Tega pa ne moremo trditi za faktor spremembe integracije. Poglejmo si nekaj enostavnih primerov. Tipično faktor IM ne dosega tako visokih vrednosti, ker so moduli veliki, z definiranimi testnimi moduli in je posledično faktor IM manjši.

1. Funkcionalnost, ki zahteva nivo spremembe 0, vzemimo certifikacijo modula na novi verziji integrirane programske opreme. V ta namen moramo ponoviti

celotno testiranje modula. Prej smo testirali modul na dveh verzijah integrirane programske opreme, sedaj bomo dodali še tretjo. Faktor spremembe integracije je zato določen na 50.

2. Funkcionalnost, ki zahteva nivo spremembe 1, vzemimo prilagoditev modula na dodatno platformo. Denimo, da obstoječi modul trenutno že deluje na štirih različnih platformah, sedaj pa ga bomo dodali še na eno. V tem primeru lahko IM določimo kot 25 odstoten. Zato, da se bo integracija testirala na vseh štirih platformah, je potrebno za 25% razširiti testiranje.

3. Funkcionalnost, ki zahteva nivo spremembe 2, predpostavimo, da je dodana samo sprememba funkcionalnosti obstoječega agenta. Denimo, da agent testiramo s 15 obstoječimi testnimi scenariji. Za to, da bomo testirali še dodano funkcionalnost, bo potrebno dodati še 3 testne scenarije. V tem primeru lahko govorimo o 20 odstotni spremembi integracije.

4. Funkcionalnost, ki zahteva nivo spremembe 3, dodali bomo novega integracijskega agenta. V tem primeru bo potrebno dodati povsem nove teste, ki bodo testirali tega integracijskega agenta. Faktor spremembe integracije je v takšnem primeru lahko postavljen na 100. Pravzaprav bi bilo pravilno v tem primeru določiti seznam testnih scenarijev, ki so potrebni, da naredimo test povprečno kompleksnega integracijskega agenta. V primeru, ko je potrebno za novo integracijo ponoviti vse teste, gre za faktor spremembe 100. V primeru, če je nova integracija bolj enostavna in lahko nekaj testnih scenarijev opustimo, lahko faktor IM temu ustrezno zmanjšamo. V primeru pa, če gre za novo bolj kompleksno integracijo, lahko faktor spremembe povečamo.

5. Funkcionalnost, ki zahteva nivo spremembe 4. Denimo, da se odločimo za arhitekturno spremembo podatkovne baze. Seveda mora ta baza zadostiti tudi kriterijem prejšnje, ker pa smo jo seveda spremenili z namenom, moramo seveda dodati še dodatne teste, ki bodo pokazali, da koncept dejansko omogoča zaželeno razširljivost, zmogljivost, ... Denimo, da bodo ti testi zahtevali še 50 odstotkov več testiranja te funkcionalnosti. V tem primeru bo faktor IM postavljen na 150.

Za razliko od prejšnjih faktorjev DM in IM, ki ne moreta biti večja kot 100, pa je faktor IM lahko večji.

4.1.6. Določitev velikosti projekta Aragon

Tabela 3 določa velikost tekočega projekta Aragon na osnovi zgoraj opisanih postopkov. Glavna pomoč pri oceni je bil seznam funkcionalnosti, zahtevan nivo spremembe za vsako funkcionalnost in matrika funkcionalnosti in modulov. $KSLOC_{EQ\_E}$

	DP55	CLI	CORE	CS	DA	DB	DR	GUI	INST	INT	MA	OTH	ZDB
KSLOC	**1453**	65.2	98.4	109.5	134.7	89.6	97.8	274.4	38.2	240.6	132.7	52.8	119.0
DM	**23.07**	5	50	30	10	20	0	20	50	30	50	5	5
CM	**15.56**	15	15	15	10	10	0	15	50	15	50	10	10
IM	**23.07**	5	50	30	10	20	0	20	50	30	50	0	5
SU	**28.91**	30	30	30	30	30	30	30	30	30	30	30	30
UNFM	**0.63**	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6	0.6
AAF	**21.06**	8.00	39.50	25.50	10.00	17.00	0.00	18.50	50.00	25.50	50.00	5.00	6.50
AAM_E	**0.29**	0.11	0.54	0.35	0.14	0.23	0.00	0.25	0.68	0.35	0.68	0.07	0.09
AAM_B	**0.21**	0.08	0.40	0.26	0.10	0.17	0.00	0.19	0.50	0.26	0.50	0.05	0.07
$KSLOC_{EQ\_E}$	**419.8**	7.1	52.8	38.0	18.3	20.7	0.0	69.0	26.0	83.4	90.3	3.6	10.5
$KSLOC_{EQ\_B}$	**308.6**	5.2	38.9	27.9	13.5	15.2	0.0	50.8	19.1	61.3	66.4	2.6	7.7

Tabela 3: Ocenjena velikost projekta Aragon

Prvi stolpec DP55 vsebuje kumulativne vrednosti za celoten produkt, ostali stolpci pa opisujejo posamezne module produkta Data Protector. Vrstice določajo:

- KSLOC – velikost modula na začetku projekta,
- DM – faktor spremembe načrta za modul,
- CM – faktor spremembe kode za modul,
- IM – faktor spremembe integracije za modul,
- SU – razumljivost programske opreme, ki je bila ocenjena na povprečno vrednost,
- UNFM – programerjevo nepoznavanje modula, ki je bilo ocenjeno na povprečno,
- AAF – utežen faktor spremembe,
- AAM_E – faktor spremembe modula, če upoštevamo SU in UNFM,

- AAM_B – faktor spremembe modula, če zanemarimo SU in UNFM,
- $KSLOC_{EQ\_E}$ – ekvivalent vrstic kode, če upoštevamo SU in UNFM,
- $KSLOC_{EQ\_B}$ – ekvivalent vrstic kode, če ne upoštevamo SU in UNFM,
- AAM – stopnja ocene in asimilacije, ki je nisem upoštevala, ker ni potrebna, ker gre za serijo projektov.

Iz analize zgornje tabele lahko zaključimo, da so na projektu Aragon spremenjeni skoraj vsi moduli, razen modula DR. Spremembe so bile večje na modulih CORE, CS, DB, GUI, INST, INT in MA. Relativno se je najbolj spremenil modul MA. Ta modul ima tudi največjo ocenjeno velikost 90.3 KSLOC. Velik delež k končni velikosti produkta prinesejo tudi moduli CORE – 52.8 KSLOC, GUI – 69 KSLOC, INT – 84.5 KSLOC.

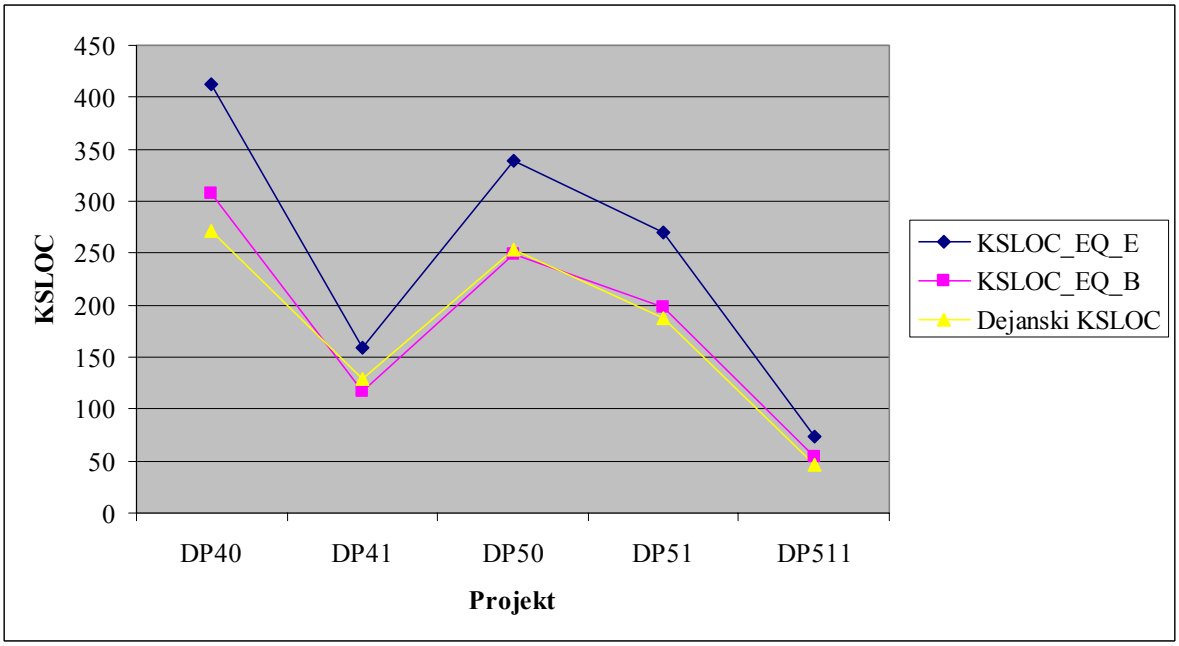
Skupna ocenjena velikost projekta Aragon je 419.8 KSLOC v primeru, če upoštevamo faktorja SU in UNFM in 308.6 KSLOC v primeru, če teh dveh faktorjev ne upoštevamo.

4.1.7. Razmerje med ekvivalentom vrstic kode in dejansko napisanimi vrsticami kode

Ko sem ocenjevala velikost projekta se mi je porajalo vprašanje, kakšno je razmerje med dejansko napisanimi vrsticami kode na projektu in ekvivalentom vrstic kode, ki ga dobimo na zgoraj opisan način. Za pretekle projekte, sem naredila podobno tabelo, le da sem tokrat upoštevala dejansko število vrstic kode, ki so bile spremenjene na projektu. Slika 6 nam prikazuje, da sta podatka o dejansko spremenjenih vrsticah kode in število vrstic kode, ki ga določimo po modelu, če ne upoštevamo faktorjev razumljivost programske opreme SU in programerjevo nepoznavanje modulov UNFM, primerljiva. Medtem, ko je v primeru, če ta faktor upoštevamo ocenjena velikost projekta večja.

Zaključimo lahko, da je dejansko število vrstic spremenjene kode dokaj dobra mera za velikost projekta, v primeru, če delajo na projektu izkušeni ljudje, ki dobro poznajo

module. V primeru, ko delajo na projektu neizkušeni ljudje, pa je ocenjena velikost projekta premajhna. Povprečno je v tem primeru ocena premajhna za 45 odstotkov.



Slika 6: Primerjava mer za velikost projekta

4.2. Ocena faktorjev eksponenta E

Naslednji korak, ki ga je potrebno narediti za parametrično oceno projekta, je določiti, kakšni so faktorji, ki vplivajo na projekt. Potrebno je določiti, kakšen je eksponent E za projekt, ki ga ocenjujemo. Na vrednost eksponenta E vplivajo faktorji, ki so opisani v razdelku 3.3.1. V nadaljevanju sledi opis faktorjev eksponenta E za tekoči projekt Aragon.

4.2.1. PREC – Poznavanje predhodnih projektov

Pri ocenjevanju vrednosti PREC nas zanima odgovor na vprašanje, v kolikšni meri je projekt podoben prejšnjemu. Ocenjujemo ujemanje projekta s prejšnjimi. Tabela 4 nam prikazuje možne opisne ocene in pripadajoče numerične vrednosti faktorja

PREC. Tabela 5 nam prikazuje skupno oceno faktorja PREC, ki je postavljena na visoko vrednost.

Opisna ocena	Vrednost PREC
Zelo nizko ujemanje projekta s prejšnjimi	6.2
Nizko ujemanje projekta s prejšnjimi	4.96
Povprečno ujemanje projekta s prejšnjimi	3.72
Visoko ujemanje projekta s prejšnjimi	2.48
Zelo visoko ujemanje projekta s prejšnjimi	1.24
Izredno visoko ujemanje projekta s prejšnjimi	0.00

Tabela 4: Tabela za oceno vrednosti PREC

1. Kako organizacija razume cilje projekta?

Organizacija dobro razume cilje projekta. Projektna organizacija je postavljena tako, da projekt gladko teče. Vsakdo ima jasno določene cilje na projektu. Zahteve projekta se na samem začetku zelo jasno definirajo v obliki dokumenta zahtev (FURPS). Določijo se tudi prioritete na projektu. Kako pomemben je čas konca projekta, kakšna je fleksibilnost glede resursov, ... Obstajajo pa tudi težave, prihaja do sprememb v projektni ekipi, delo določenega dela ekipe bo premaknjeno na drugo lokacijo. Zato je ocena razumevanje ciljev na projektu ocenjeno kot povprečno.

Lastnost	Opisna vrednost
Kako organizacija razume cilje projekta	Povprečna
Izkušnje pri delu s podobnimi projekti	Visoka
Hkraten razvoj nove strojne opreme	Zelo visoka
Potreba po novi arhitekturi, algoritmih	Visoka
PREC	**Visoka**

Tabela 5: Ocene PREC za Aragon projekt

2. Kolikšne so izkušnje pri delu s podobnimi projekti?

Projekt je še eden v verigi obstoječih programskih projektov v okviru razvoja programskega produkta Data Protector. Projektna ekipa ima s tovrstnimi produkti zadosti izkušenj. Znotraj vsakega od projektov se izvede vsaj nekaj izboljšav na projektu na osnovi izkušenj s prejšnjih projektov. Kljub temu pa prihaja na tem projektu do določenih sprememb na področjih, kjer še nimamo izkušenj. Predvsem,

kar se tiče organizacije tima. Izkušnje pri delu s podobnimi projekti so zato ocenjene na oceno visoke.

3. Ali poteka hkrati tudi razvoj nove strojne opreme?

Za projekt Aragon ne poteka hkraten razvoj strojne opreme. V primeru, ko na projektu ni novega razvoja strojne opreme, je ocenjena vrednost zelo visoka.

4. Ali se razvijajo novi inovativni algoritmi?

Na projektu Aragon se razvijajo novi algoritmi za shranjevanje kopije podatkov. V okviru te funkcionalnosti se razvijajo novi algoritmi, ki bodo bistveno izboljšali koncepte shranjevanja kopije podatkov. V primeru, če se razvija na projektu nekaj novih algoritmov je ocenjena vrednost visoka.

4.2.2. FLEX – Prilagodljivost razvoja

Pri ocenjevanju prilagodljivosti razvoja skušamo poiskati odgovor na vprašanje, do kakšne mere se mora razvoj prilagoditi nekim postavljenim zahtevam. Tabela 6 nam prikazuje opisne ocene in pripadajoče numerične vrednosti faktorja FLEX, Tabela 7 pa skupno vrednost faktorja FLEX, ki je za projekt Aragon postavljena na povprečno oceno.

Opisna ocena	Vrednost FLEX
Zelo nizka, strogo zahtevana skladnost	5.07
Nizka, nizka prilagodljivost razvoja	4.05
Povprečna, nekaj prilagodljivosti v razvoju	3.04
Visoka, splošna skladnost	2.03
Zelo visoka, nekaj skladnosti	1.01
Izredno visoka, splošno zahtevani cilji	0.00

Tabela 6: Tabela za oceno vrednosti FLEX

1. Skladnost programske opreme s predhodno določenimi zahtevami

Ker gre za nadaljevanje razvoja že obstoječega programskega produkta, je kar nekaj zahtev glede programske opreme že postavljenih. Razvoj nove funkcionalnosti mora konsistentno dopolnjevati že obstoječo funkcionalnost. Kljub temu pa obstaja nekaj prilagodljivosti v razvoju. V primeru, če je ocenjena potreba po bolj koreniti spremembi, se tudi lahko dogovorimo za takšno spremembo. Tako je FLEX za ta kriterij ocenjen na povprečen.

2. Skladnost programske opreme z eksternimi vmesniki

Programski vmesniki za produkt so že razviti in se v okviru trenutnega projekta samo prilagajajo funkcionalnostim, ki se dodajajo v produkt. Zaželena je čim večja skladnost z obstoječimi eksternimi vmesniki. Ocena FLEX je ocenjena na nizko.

3. Zahtevana skladnost in hkrati hiter konec projekta

Zahteva po hitrem koncu projekta Aragon se je skozi projekt spreminjala. Na samem začetku je izgledalo, da je čas pomemben, potem pa se je zaradi drugih vzrokov pomembnost časa začela zmanjševati, proti koncu projekta pa je zopet naraščala. Pri tem kriteriju pa v bistvu ocenjujemo kombinacijo nefleksibilnosti zahtev po skladnosti in zahteve po hitrem koncu projekta. To bi ocenila na nizko, čemur ustreza faktor FLEX z visoko vrednostjo.

Lastnost	**Opisna vrednost**
Zahtevana skladnost programske opreme s predhodno določenimi zahtevami.	Povprečna
Zahtevana skladnost z eksternimi vmesniki.	Nizka
Zgornje zahteve v kombinaciji z zahtevo po hitrem koncu projekta.	Visoka
PREC	**Povprečna**

Tabela 7: Ocene FLEX za Aragon projekt

Na zgornja dva faktorja pri samem projektu težko vplivamo in ju v bistvu le ocenimo. Pri naslednjih treh faktorjih pa lahko s posredovanjem pri vodenju zmanjšamo neracionalnosti na projektu in na takšen način pripomoremo k bolj učinkovitim projektom.

4.2.3. RESL – Arhitektura in reševanje kritičnih situacij

Pri tem faktorju ocenimo, kako se na projektu rešujejo arhitekturni problemi in kritične situacije. Ocenimo odstotek situacij, kjer se ti problemi ustrezno rešujejo. Tabela 8 nam prikazuje opisne ocene in pripadajoče numerične vrednosti za faktor RESL, Tabela 9 pa skupno vrednost faktorja RESL, ki je ocenjena na povprečno.

Opisna ocena	Vrednost RESL
Zelo nizka, zelo malo problemov se ustrezno rešuje (20%)	7.07
Nizka, nekaj problemov se ustrezno rešuje (40%)	5.65
Povprečna, problemi se pogosto ustrezno rešujejo (60%)	4.24
Visoka, problemi se v splošnem ustrezno rešujejo (75%)	2.83
Zelo visoka, problemi se večinoma ustrezno rešujejo (90%)	1.41
Izredno visoka, problemi se v vseh primerih ustrezno rešujejo (100%)	0.00

Tabela 8: Tabela za oceno vrednosti RESL

1. Plan reševanja kritičnih situacij je skladen z življenjskim ciklom projekta

Plan kritičnih situacij na projektu obstaja in se redno prilagaja trenutnemu stanju na projektu. Za projekt se po rednih tedenskih sestankih, glede na trenutno situacijo, pripravi seznam kritičnih situacij, ki se tedensko tudi preveri na rednih sestankih upravljavske ekipe. Za vsakega od faktorjev tveganja je definiran tudi načrt, kako nadaljevati, v primeru da se tveganje udejanji, obenem pa se določi tudi pogoj, ki se mora izpolniti, da sprožimo pripravljeni načrt. Zato je ocena plana reševanja kritičnih situacij postavljena na zelo visoko vrednost.

2. Plani na projektu so skladni s planom reševanja kritičnih situacij

Ostali načrti na projektu, kot so načrt dela, proračun in interni mejniki v razvoju programske opreme, se skladajo s planom reševanja kritičnih situacij. Načrti na projektu se glede na trenutno situacijo in kritične situacije, ki so se med projektom

uresničile, prilagajajo. Na koncu vsake od faz življenjskega cikla se pogleda tudi skladnost vseh pripravljenih planov. Projekt ne more v naslednjo fazo, če plani niso skladni. Res pa je, da so faze glede na velikost projekta lahko dolge in da se formalno ne pregleduje, če so načrti ves čas skladni s planom kritičnih situacije. Tako je ocena plana reševanja kritičnih situacij postavljena na visoko.

3. Odstotek časa rezerviran za arhitekturo produkta

Odvisno od seznama funkcionalnosti, ki se dodaja na novem projektu, se tipično prilagodi tudi odstotek časa, ki je namenjen za arhitekturo projekta. Za vse funkcionalnosti je bilo potrebno pripraviti tako eksterno specifikacijo, kot tudi načrt, če le to zahtevajo. Odstotek časa, ki je namenjen načrtu, je ocenjen na približno 10 odstotkov časa na projektu, kar bi lahko ocenili z nizko oceno.

4. Odstotek arhitektov, ki so nam na voljo, glede na potrebe

V primeru, da se spreminja načrt produkta, se v spremembe vključijo tako vodilni inženirji, kot tudi arhitekti. Včasih bi bila potrebna še večja vpletenost arhitektov, ki bi skrbeli zato, da bi projekt ostal konsistenten. Ne obstaja formalen mehanizem, ki bi zahteval, da se vsak načrt, oziroma eksterna specifikacija pregleda tudi s strani arhitekta. Tako je ocena za vključenost arhitektov ocenjena na nizko.

5. Orodja, ki so nam na voljo za reševanja kritičnih situacij razvoj in verifikacijo produkta

V primeru reševanja kritičnih situacij nimamo na voljo posebnih orodij, ki bi nam pomagala pri razrešitvi le-teh. Prav tako nimamo ustreznih orodij za razvoj specifikacij za načrt in za verifikacijo arhitekture. Ocena je postavljena na zelo nizko.

6. Stopnja negotovosti za ključne dele arhitekture produkta

Stopnja negotovosti za ključne dele programskega produkta kot so strojna oprema, zmogljivost in uporabniški vmesnik, ni preveč velika. Za vse funkcionalnosti se pripravi na samem začetku projekta tako eksterna specifikacija, kot tudi načrt. V primeru, da nismo popolnoma prepričani v predlagamo rešitev, se naredi še

predhodna raziskava, ki nam odpravi morebitne nejasnosti. Za projekt Aragon se kaže še najbolj negotovo samo testiranje produkta, ker prihaja do sprememb v razporejanju nalog drugim ekipam na drugih lokacijah. Ker pa obstaja nekaj negotovosti, je ocena postavljena na povprečno.

7. Število in stopnja zahtevnosti kritičnih situacij

Na projektu so povprečno v tabeli kritičnih situacij 4, ki so označene kot kritične. Ta vrednost določa vrednost RESL kot povprečno.

Lastnost	**Opisna vrednost**
Plan reševanja kritičnih situacij je skladen z življenjskim ciklom projekta	Zelo visoka
Plani na projektu so skladni s planom reševanja kritičnih situacij	Visoka
Odstotek časa rezerviran za arhitekturo produkta	Nizka
Odstotek arhitektov, ki so nam na voljo, glede na potrebe	Povprečna
Orodja, ki so nam na voljo za reševanja kritičnih situacij razvoj in verifikacijo produkta	Zelo nizka
Stopnja negotovosti za ključne dele arhitekture produkta	Povprečna
Število in stopnja zahtevnosti kritičnih situacij	Povprečna
RESL	**Povprečna**

Tabela 9: Ocene RESL za projekt Aragon

4.2.4. TEAM – Skladnost različnih projektnih skupin

V tem primeru ocenjujemo, kako skladno sodelujejo na projektu nosilci različnih vlog, ki zastopajo vsak svoje interese: uporabniki, naročniki, skupine za podporo, razvijalci in drugi. V bistvu ocenjujemo odgovor na vprašanje, kako poteka medsebojna komunikacija med le-temi skupinami. Lestvica vrednosti za faktor TEAM opisana v tabeli Tabela 10.

Opisna ocena	**Vrednost TEAM**
Zelo nizka, komunikacija izredno težko poteka	5.48
Nizka, komunikacija včasih težko poteka	4.38
Povprečna, v splošnem je komunikacija ustrezna	3.29
Visoka, komunikacija je zelo dobra	2.19
Zelo visoka, komunikacija je izredno dobra	1.10
Izredno visoka, komunikacija poteka popolnoma brez problemov	0.00

Tabela 10: Tabela za oceno vrednosti TEAM

1. Konsistentnost ciljev skupin v različnih vlogah na projektu

Projekt je izredno velik. Med seboj morajo sodelovati različne skupine. Skupine lahko delimo v skupino naročnikov, skupin vodij na projektu, skupino razvijalcev, skupino testerjev, skupino, ki skrbi za uporabniško dokumentacijo in skupino za podporo. V bistvu komunikacija dobro poteka na ravneh, kjer so procesi že jasno določeni. V primeru, da procesov zaradi nove narave problemov še ni, pa je komuniciranje težje. Težave v komunikaciji nastanejo tudi v primeru, ko je potrebna komunikacija timov, ki se nahajajo na različnih lokacijah. V tem primeru, poteka komunikacija počasneje. Ker se bo del testiranja odvijal na oddaljeni lokaciji, s katero do sedaj nimamo še nobenih izkušenj, bi ta faktor ocenila kot povprečen.

2. Pripravljenost prilagajanja med skupinami

V primeru problemov se skupine medsebojno prilagajajo. Vedno se poišče način, da se problem razreši. V primeru, da procesi še ne obstajajo se le-ti ustrezno definirajo, v primeru, če menimo, da bo do podobnih situacij še prihajalo. Mislim, pa da bo prilagajanje v QA skupini na tem projektu oteženo, zato je ta faktor ocenjen kot povprečen.

3. Izkušnje skupin pri skupnem delu

Izkušnje pri skupnem delu so visoke, vendar pa imamo na projektih tudi začetnike, pa tudi sama struktura skupin se spreminja med projekti. Lahko pa ocenimo, da vedno dela na projektu dobršen odstotek ljudi, ki imajo za seboj že kar nekaj podobnih projektov. Ocena je postavljena na visoko.

4. Druženje skupin

Različne skupine na projektu se družijo med seboj. Ravno v času projekta je bil organiziran eden takšnih dogodkov. V tem primeru, pa je šlo bolj za druženje največje skupine, ki dela na eni od lokacij. Ocena je postavljena na povprečno.

Lastnost	Opisna vrednost
Konsistentnost ciljev skupin v različnih vlogah	Povprečna
Pripravljenost prilagajanja med skupinami	Povprečna
Izkušnje skupin pri skupnem delu	Visoka
Druženje skupin	Povprečna
TEAM	**Povprečna**

Tabela 11: Ocena vrednosti TEAM za Aragon

4.2.5. PMAT – Zrelost procesov

Lestvica ocen, ki je definirana za FAKTOR PMAT je naslednja:

1. CMM nivo 1, spodnja polovica, zelo nizka - 7.08
2. CMM nivo 1, zgornja polovica, nizka - 6.24
3. CMM nivo 2, povprečna - 4.68
4. CMM nivo 3, visoka - 3.12
5. CMM nivo 4, zelo visoka - 1.56
6. CMM nivo 5, izredno visoka - 0

Ocena podjetja HSL po SEI CMM lestvici je nivo 2, kar pomeni oceno 4.68.

4.2.6. Vsota faktorjev eksponenta

Tabela 12 nam prikazuje končne ocene faktorjev eksponenta in njihovo vsoto.

Lastnost	Opisna ocena	Numerična ocena
PREC	Visoka	2.48
FLEX	Povprečna	3.04
RESL	Povprečna	4.24
TEAM	Povprečna	3.29
PMAT	Povprečna	4.68
Vsota faktorjev eksponenta		**17.73**

Tabela 12: Vsota faktorjev eksponenta za projekt Aragon

4.3. Ocena faktorjev dela

V tem razdelku so opisane ocene faktorjev dela za programski projekt Aragon. Kljub temu, da se načeloma faktorji dela ocenjujejo glede na modul, pa sem jih v nadaljevanju ocenila glede na produkt. Gre za prvi poizkus uporabe modela, na katerem sem delala samostojno. V primeru, da se odločimo, da model uporabimo kot pomoč pri ocenjevanju, pa bi naredili še naslednji korak in bi faktorje dela ocenili glede na modul. Med faktorje dela sodijo faktorji, ki so opisani v razdelku 3.3.2.

4.3.1. Faktorji produkta

1. Zahtevana zanesljivost programske opreme (RELY)

V tabeli spodaj so opisane težave, ki nas lahko doletijo v primeru napak na produktu. Za projekt Aragon bi tveganje ob napakah ocenila kot povprečno. Nevarnost tveganja sicer obstaja. V primeru, da je v produktu preveč napak, potem stranke lahko izgubijo zaupanje v produkt in se odločijo za konkurenčni produkt.

Opis težav zaradi napake	Opisna ocena	Faktor
Manjše neprijetnosti	Zelo nizka	0.82
Lahko popravljive napake	Nizka	0.92
Načeloma lahko popravljive napake	Povprečna	1
Visoka finančna izguba	Visoka	1.10
Nevarnost izgube človeških življenj	Zelo visoka	1.26

Tabela 13: Faktor RELY

2. Velikost podatkovne baze (DATA)

V primeru programskega produkta Data Protector poudarjamo testiranje velikih podatkovnih baz v primeru, da se večje spremembe dejansko dogajajo na baznem modulu. Za ta produkt tovrstnih sprememb ni bilo veliko, tako da je ocena postavljena na povprečno, ki v končni fazi ne vpliva na samo potrebno delo na projektu.

3. Kompleksnost produkta (CPLX)

Faktor pokriva 5 področij, ki se ocenijo glede na posebne kriterije s šestimi različnimi opisnimi ocenami. Tabela 14 prikazuje tudi numerične faktorje, ki pripadajo opisnim ocenam.

Opis kompleksnosti	Opisna ocena	Faktor
Izredno preproste operacije	Zelo nizka	0.73
Preproste operacije	Nizka	0.87
Povprečno kompleksne operacije	Povprečna	1.00
Kompleksne operacije	Visoka	1.17
Zelo kompleksne operacije	Zelo visoka	1.34
Izredno kompleksne operacije	Izredno visoka	1.74

Tabela 14: Faktor CPLX

1. kontrolne operacije – kompleksnost kontrolnih operacij za projekt Aragon je ocenjena na povprečno. Bistvo produkta za shranjevanje podatkov so tudi kontrolne operacije, ki pa so načeloma preproste.

2. računske operacije – kompleksnost računskih operacij je ocenjena na nizko. Produkt za shranjevanje podatkov ne potrebuje zelo kompleksnih operacij. V večini modulov gre za manj kompleksne računske operacije.

3. operacije odvisne od naprav – kompleksnost operacij z napravami je ocenjena na visoko. Produkt za shranjevanje podatkov mora podpirati krmiljenje naprav na fizičnem nivoju.

4. operacije za upravljanje s podatki – operacije za upravljanje s podatki na projektu so ocenjene kot visoke. Potrebne so tudi spremembe na podatkovni bazi.

5. operacije pri upravljanju z uporabniškimi vmesniki – operacije za upravljanje uporabniških vmesnikov so ocenjene na povprečne. Produkt ima razvit grafični vmesnik, ki se znotraj projekta le razširja glede na zahteve nove funkcionalnosti.

Skupna ocena faktorja CPLX je postavljena na povprečno.

Tip operacij	Opisna ocena	Faktor
Kontrolne operacije	Povprečna	1
Računske operacije	Nizka	0.87
Operacije za krmiljenje naprav	Visoka	1.17
Operacije za upravljanje s podatki	Visoka	1.17
Operacije za uporabniški vmesnik	Povprečna	1
Skupaj	Povprečna	1.04

Tabela 15: Ocena faktorja CPLX za Aragon

4. Razvijanje produkta za ponovno uporabo (RUSE)

Produkt projekta Aragon je nova verzija programskega projekta Data Protector. Zato je faktor razvijanja produkta za ponovno uporabo temu primerno visok. Glede na tabelo spodaj je ocenjen na zelo visoko vrednost, ki se določi v primeru, ko projekti razvijajo nove verzije produkta na produktni liniji.

Opis ponovne uporabe	Opisna ocena	Faktor
	Zelo nizka	Ni ocene
Ne obstaja	Nizka	0.95
Na projektu	Povprečna	1
Na programu	Visoka	1.07
Na produktni liniji	Zelo visoka	1.15
Med produktnimi linijami	Izredno visoka	1.24

Tabela 16: Faktor RUSE

5. Ustreznost dokumentacije trenutnemu stanju procesa (DOCU)

Za projekt Aragon imamo zahtevo, da se dokumentacija primerno sklada s trenutnim življenjskim ciklom projekta. Ni pa kakšnih posebnih zahtev glede tega. Uporabniška dokumentacija ima definirane svoje mejnike v razvoju, ki pa se skladajo z celotnim razvojem na projektu.

6. Skupna ocena faktorjev produkta

V tabeli spodaj lahko vidimo skupno oceno faktorjev dela.

Faktorji dela	Opisna ocena za Aragon	Faktor
RELY	Povprečna	1
DATA	Povprečna	1
CPLX	Povprečna	1.04
RUSE	Zelo visoka	1.15
DOCU	Povprečna	1
Skupaj		1.20

Tabela 17: Faktorji produkta ocenjeni za Aragon

4.3.2. Faktorji platform

1. Omejitev časa izvajanja programa (TIME)

Gre za oceno odstotka razpoložljivega časa izvajanja programa, ki ga nameravamo izkoristiti na projektu z danimi resursi. Glede na to, da je programski produkt Data Protector zelo kompleksen, se naredi načrt testiranja, ki pokrije samo nek odstotek možnih testnih scenarijev. Ocena je zato postavljena na povprečno.

2. Omejitev glavnega pomnilnika (STOR)

Tudi ocena porabe glavnega pomnilnika na produktu Data Protector je postavljena na povprečno.

3. Stabilnost platforme (PVOL)

Produkt Data Protector teče na različnih platformah. Vedno pa se pojavljajo zahteve po še novih platformah. Tudi na tem področju lahko ocenimo spremembe kot nominalne, torej večja sprememba dvakrat na leto in manjša sprememba dvakrat na mesec (Tabela 18).

Ocena sprememb	Opisna ocena	Faktor
	Zelo nizka	Ni definiran
Večja sprememba vsako leto, manjša sprememba vsak mesec	Nizka	0.87
Večja sprememba dvakrat na leto, manjša sprememba dvakrat na mesec	Povprečna	1
Večja sprememba na 2 meseca, manjša sprememba vsak teden	Visoka	1.15
Večja sprememba na 2 tedna, manjša sprememba na 2 dni	Zelo visoka	1.30
	Izredno visoka	Ni definiran

Tabela 18: Faktor PVOL

4. Skupna ocena faktorjev platforme

Glede na zgoraj podane ocene, lahko faktor platforme za projekt Aragon opustimo, ker je ocenjen na 1.

Faktorji platforme	Opisna ocena za Aragon	Faktor
TIME	Povprečna	1
STOR	Povprečna	1
PVOL	Povprečna	1
Skupaj		1

Tabela 19: Faktorji platforme ocenjeni za projekt Aragon

4.3.3. Faktorji osebja

1. Sposobnost analitika (ACAP)

Ker nimamo mer, kako določiti izkušenost analitika, sem jo postavila na povprečno.

2. Sposobnost programerja (PCAP)

Ker nimamo mer, kako določiti izkušenost programerja, sem jo postavila na povprečno.

3. Stalnost osebja (PCON)

Projekt Aragon je bil zaradi poslovnih odločitev izpostavljen večji fluktuaciji osebja, zato je ocena stalnosti osebja tudi ocenjena na povprečno.

4. Izkušnje z aplikacijo (APEX)

Izkušnje z aplikacijo so ocenjene na visoko. Povprečno imajo inženirji na projektu tri leta izkušenj s projektom.

5. Izkušnje s platformo (PLEX)

Platforma na kateri se produkt razvija postaja sicer bolj kompleksna, ker se produkt prilagaja na različne nove platforme, vendar pa imajo inženirji s platformo na kateri razvijajo produkt povprečno vsaj 3 letne izkušnje. Zato so izkušnje s platformo ocenjene na zelo visoko.

6. Izkušnje s programskim jezikom in orodji (LTEX)

Inženirji so dobro seznanjeni tudi z orodji, ki se uporabljajo na projektu. Orodja se ne spreminjajo bistveno, tako da imajo ravno tako povprečno 3 leta izkušenj pri delu z njimi. Ocenjena je na zelo visoko.

7. Skupna ocena faktorjev osebja

V tabeli spodaj so vpisane posamezne ocene za faktor osebja na projektu Aragon in pa skupna ocena.

Faktorji osebja	Opisna ocena za Aragon	Faktor
PCAP	Povprečna	1
PCON	Povprečna	1
APEX	Visoka	0.88
PLEX	Zelo visoka	0.85
LTEX	Zelo visoka	0.84
Skupaj		0.91

Tabela 20: Faktorji osebja ocenjeni za projekt Aragon

4.3.4. Faktorji projekta

1. Uporaba programskih orodji (TOOL)

Programska orodja razvijalce so odvisna od platforme, na kateri razvijajo. Na Microsoft platformah se uporablja Developers Studio, na UX okolju pa make. Za upravljanje konfiguracije se uporablja programsko orodje Clear Case, ki je še nadgrajeno z aplikacijo za lažje sledenje spremembam med posameznimi verzijami dnevno prevedene kode. Testna ekipa uporablja posebno aplikacijo, ki ji omogoča lažje shranjevanje testnih skript, definicijo množice testnih programov, ki se morajo izvesti v posameznem ciklu projekta, sledenje testnih planov, ... Za lažje sledenje nepravilnosti in napak na produktu se problemi, ki se najdejo pri testiranju shranjujejo v bazo napak (DDTS - Distributed Defect Tracking System), kjer se potem tudi sledijo. Za planiranje in sledenje projektu se uporablja MS Project. Inženirji, ki pišejo uporabniško dokumentacijo uporabljajo Frame Maker.

Skupna ocena orodij, ki jih uporabljamo na projektu, je glede na kriterije modela postavljena na visoko. Orodja, ki se uporabljajo so profesionalna, niso pa zadosti dobro povezana med seboj.

2. Razvoj na več oddaljenih lokacijah (SITE)

Produkt se je do sedaj razvijal na dveh lokacijah, večji del v Ljubljani, drugi pa v podjetju HP v Nemčiji. Dodatno se je vključila še tretja skupina, ki dela v Indiji. Tako je ocena postavljena na nizko, saj gre tako za sodelovanje z različnimi mesti, kot tudi z različnima podjetji.

3. Zahtevan plan razvoja (SCED)

Projekt bo zaradi zmanjševanja število inženirjev trajal dlje kot običajno. Faktor SCED je tako postavljen na vrednost visoko, kar je 130% nominalne vrednosti. Ta vrednost pa ne vpliva na delo, ker je večja od nominalne, temveč le na dolžino projekta.

4. Skupna ocena faktorjev projekta

V tabeli spodaj so vpisane posamezne ocene za faktor osebja na projektu Aragon in pa skupna ocena.

Faktorji projekta	Opisna ocena za Aragon	Faktor
TOOL	Visoka	0.90
SITE	Nizka	1.09
SCED	Visoka	1.00
Skupaj		1.00

Tabela 21: Faktorji projekta ocenjeni za produkta Aragon

4.3.5. Povzetek ocen faktorjev dela za projekt Aragon

Faktorji dela	Faktor
Faktorji produkta	1.20
Faktorji platforme	1.00
Faktorji osebja	0.91
Faktorji projekta	1.00

Tabela 22: Ocene faktorjev dela za projekt Aragon

Produkt faktorjev projekta za produkt Aragon je ocenjen na povprečnega. Pozitiven vpliv faktorjev osebja se izniči z negativnim vplivom faktorjev produkta.

4.4. Preslikava življenjskega cikla projekta

COCOMO II model podpira linearen model življenjskega cikla projekta. Prav tako pa tudi pri razvoju programske opreme za produkt Data Protector uporabljamo linearen model življenjskega cikla projekta. Seveda obstajajo med obema modeloma pomembne razlike. V nadaljevanju sledi opis življenjskega cikla projekta, ki ga predpostavlja COCOMO II in življenjskega cikla projekta, ki se uporablja na Data Protector projektih, ter primerjava oziroma ustrezna preslikava med obema življenjskima cikloma.

4.4.1. Linearen model življenjskega cikla uporabljen v modelu COCOMO II

Linearen model življenjskega cikla projekta, ki se predpostavlja v primeru uporabe COCOMO II modela, je sestavljen iz kontrolnih točk projekta in zahtev, ki morajo biti izpolnjene na vsaki od kontrolnih točk, ki so opisane v naslednjih razdelkih.

1. Začetek faze načrtovanja in zahtev

Na začetku faze načrtovanja in zahtev mora biti določen in pregledan koncept življenjskega cikla projekta (LCR - Life Cycle Concept Review). Poleg tega pa morajo biti izpolnjeni še naslednji pogoji:

- Potrjena arhitektura sistema, ki vključuje plan potrebne programske in strojne opreme za izvedbo projekta.
- Potrjen plan operacije, ki vključuje plan sestave projektnega tima.
- Potrjen plan življenjskega cikla projekta, ki vključuje kontrolne točke projekta, vire, zadolžitve, terminski plan in glavne aktivnosti na projektu.

2. Konec faze načrtovanje in zahtev, začetek faze načrtovanja produkta

Ob koncu faze načrtovanje in zahtev morajo biti določene in pregledane zahteve projekta (SRR - Software Requirements Review). Poleg tega morajo biti izpolnjeni naslednji pogoji:

- Določen podroben razvojni plan, določeni kriteriji prehajanja med fazami in proračun projekta.
- Določen podroben plan uporabe - plan treningov, namestitve, operacije in vzdrževanja za produkt.
- Določen podroben plan kontrole za projekt – plan za upravljanje konfiguracije, plan za zagotavljanje kvalitete.

- Potrjene specifikacije za projektne zahteve – funkcionalnost, zmogljivost, vmesniki in testiranje.
- Potrjena pogodba projekta na osnovi zgornjih elementov.

3. Konec faze načrtovanja projekt, začetek faze podrobnega načrtovanja

Glavni pogoj za konec faze načrtovanja je pregledan načrt produkta (PDR - Product Design Review), poleg tega pa morajo biti izpolnjeni še naslednji pogoji:

- Potrjena specifikacija načrta.
- Potrjene specifikacije v smislu popolnosti in konsistentnosti.
- Ugotovljene in razrešene vse kritične situacije, ki so povezane z visokim tveganjem.
- Začetni integracijski plan, testni plan, plan prevzema produkta in uporabniška dokumentacija.

4. Konec faze podrobnega načrtovanja, začetek faze kodiranja in testiranja posameznih enot

Ob koncu faze podrobnega načrtovanja mora biti pregledan podroben načrt projekta (CDR - Critical Design Review), dodatno pa morajo biti izpolnjeni še naslednji pogoji:

- Pregledan natančen načrt za vsako posamezno enoto.
- Za vsako proceduro znotraj enote (manj kot 100 vrstic) je potrebno določiti ime, namen, velikost, način klicanja, vrnjene kode, vhod, izhod, algoritem, način poteka.
- Opis podatkovne baze.
- Pregledana popolnost, konsistentnost, sledljivost zahtev in načrta sistema.
- Sprejet plan prevzema.

- Končan prvi osnutek integracijskega in testnega plana ter uporabniške dokumentacije.

5. Končano kodiranje in testiranje posameznih enot, začetek integracijske in testne faze

Ob koncu kodirana in testiranja posameznih enot morajo biti doseženi cilji, ki so bili postavljeni pri testiranju enot (UTC – Unit Test Criteria), poleg tega pa morajo biti izpolnjeni še naslednji pogoji:

- Pregledati je potrebno vsa izvajanja testiranja enote, uporabljati je potrebno tako pričakovane, kot tudi robne vrednosti.
- Pregledati je potrebno tako vhodne, izhodne parametre testiranja enot, kot tudi sporočila v primeru napak.
- Pregledati je potrebno skladnosti kode z zadanimi standardi.

6. Končana integracijska in testna faza, začetek faze implementacije

Ob koncu integracijske faze morajo biti doseženi cilji za prevzem programske opreme (SAR – Software Acceptance Review). Izpolnjeni pa morajo biti tudi naslednji pogoji:

- Doseženi cilji, ki so bili določeni za prevzem programske opreme.
- Narejen pregled, če so bile zahteve v specifikacijah dosežene.
- Pripravljena demonstracija primernih performans.
- Pripravljen mora biti prevzem vseh izdelkov te faze: poročila, uporabniška dokumentacija, specifikacije, ...

7. Končana faza implementacije, začetek faze uporabe in vzdrževanja

Ko je končana faza implementacije, morajo biti doseženi cilji za prevzem sistema (SAR – System Acceptance Review). Doseženi pa morajo biti tudi naslednji pogoji:

- Zadovoljivo narejen test za prevzem sistema.
- Narejen pregled, če so bile zahteve, ki so bile postavljena za delovanje sistema, dosežene.
- Narejen pregled, če je sistem pripravljen na uporabo, tako z vidika programske opreme, strojne opreme, dokumentacije in treningov.
- Dokončane morajo biti vse potrebne aktivnosti glede namestitve sistema.

8. Konec faze uporabe in vzdrževanja

- Dokončane vse točke, ki so potrebne v primeru, ko gre sistem iz uporabe: dokumentacija, arhiviranje, prehod na novi sistem.

4.4.2. Linearen model življenjskega cikla projekta na projektih razvoja produkta Data Protector

Linearen razvoj projekta, ki se uporablja pri razvoju produkta Data Protector II, je ravno tako sestavljen iz več faz, med njimi pa imamo definirane kontrolne točke, kjer se preveri, če so bile znotraj posamezne faze izpolnjene vse zahteve.

1. Začetek projekta

Začetek projekta, ki se začne s uvodnim sestankom (Kick Off), na katerem se definirajo cilji projekta, potrebne raziskave, projektne omejitve, projektna organizacija, pristop k razvoju na projektu in začetni terminski plan.

2. Konec faze definicije zahtev

Cilj faze 1 je definicija zahtev. Ob koncu faze 1 se sprejme dokument zahtev (FURPS), ki opisuje zahteve projekta z vidika dodane funkcionalnosti, uporabnosti, zanesljivosti, performans in vzdrževanja. Dodatno morajo biti izpolnjeni naslednji pogoji:

- Potrjen plan kvalitete.
- Potrjen dokument zahtev (FURPS).
- Potrjeni cilji projekta.
- Pregledane začetne raziskave na projektu.
- Podana prva ocena za projektno ekipo, potrebno programsko in strojno opremo in prvi terminski plan, ki pa je še zelo grob.

3. Konec faze specifikacije zahtev

Glavni cilj faze 2 je specifikacija zahtev. Na koncu faze 2 mora biti sprejet dokument, ki definira eksterno specifikacijo produkta (ERS – External Reference Specification), podroben terminski plan projekta. Izpolnjeni morajo biti naslednji pogoji:

- Sprejeta eksterna specifikacija zahtev.
- Definirana tabela za spremljanje aktivnosti na projektu.
- Napisan načrt (HLD), za funkcionalnosti kjer se je zahteval.
- Definirana matrika kritičnih situacij.
- Sprejet plan razvoja.
- Definiran plan testiranja, skupaj s planom testiranja nove funkcionalnosti.
- Definiran plan za izdelavo uporabniške dokumentacije.
- Definiran plan za upravljanja konfiguracije.

4. Konec faze implementacije

Cilj faze 3 je dokončati načrte, kodiranje, integracijsko in del sistemskega testiranja produkta. Rezultat faze 3 so naslednji dokumenti, specifikacije, poročila in programi:

- Podroben načrt novih funkcionalnosti.
- Testni scenariji za novo funkcionalnost.
- Izvorna koda.
- Poročila o pregledih izvorne kode.
- Uporabniška dokumentacija.
- Programski paketi, ki so primerni za namestitev (beta kvaliteta).
- Opisani rezultati uporabniških testnih scenarijev.

5. Konec faze prevzema

Cilj faze 4 je prevzem produkta. Rezultat faze 4 so naslednji dokumenti, specifikacije, poročila in programi:

- Preglednica za spremljanje aktivnosti na projektu.
- Paketi programskega produkta.
- Uporabniška dokumentacija.
- Matrika, ki določa konfiguracije, na katerih je posamezna funkcionalnost podprta.
- Dopolnjena interna dokumentacija.
- Poročilo o sistemskem testiranju in beta testiranju.
- Skupno poročilo o testiranju na projektu.
- Poročilo ekipe za podporo, ki je pregledala produkt.

6. Konec projekta

Konce faze 5, cilj katere je zapreti projekt. Produkt prevzame ekipa, ki dela na vzdrževanju. Na strani razvoja pa se projekt zapre. Rezultat faze 5 so naslednje komponente:

- Arhiv projekta na CD.
- DDTS baza.
- Testno okolje in testna orodja.
- Izvorna koda in okolje za prevajanje.
- Projektna statistika.

4.4.3. Primerjava obeh linearnih modelov

Tabela 23 nam prikazuje preslikavo posameznih faz linearnega modela COCOMO II in linearnega modela, ki se uporablja na projektih razvoja programskega produkta Data Protector. Vidimo lahko, da ima COCOMO II več faz z bolj jasno določenimi kontrolnimi točkami med posameznimi fazami. Bistvena razlika je opazna v fazi 3 modela Data Protector, v kateri se implementira produkt. Ta faza je v modelu COCOMO II razdeljena na 3 faze:

1. Fazo podrobnega načrtovanja.
2. Fazo kodiranja in testiranja enot.
3. Fazo integracijskega testiranja in testiranja sistema.

Tudi v primeru razvoja Data Protector projektov imamo vse te podfaze, vendar pa kontrolne točke med posameznimi fazami niso tako jasno določene, kot v primeru modela COCOMO II.

Zaključimo lahko tudi, da faza prevzema produkta, tako kot je definirana na projektih Data Protector, dejansko sega čez dve fazi COCOMO II modela. V okviru te faze se dokonča testiranje sistema in se naredi tudi testiranje, katerega namen je ugotoviti, če je produkt pripravljen na prevzem.

Faza uporabe in vzdrževanja pa ni del razvojnega projekta Data Protector. S to fazo se ukvarja ekipa, ki dela na vzdrževanju sistemov.

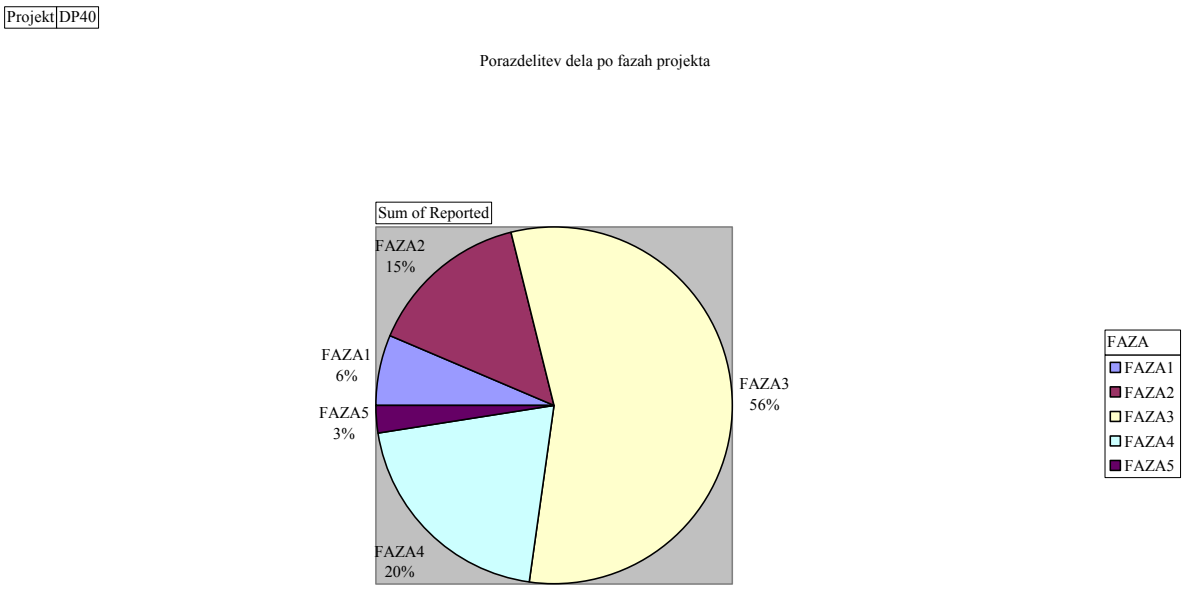
Zaključimo lahko, da sta si modela podobna, pa kljub temu dovolj različna, da ne moremo pričakovati enako količino potrebnega dela, za to da zadostimo kriterijem posamezne faze. Za to, da bo model COCOMO II primeren tudi za ocenjevanje Data Protector projektov, je potrebno opraviti še en korak, to je kalibriranje modela na osnovi lokalnih podatkov.

COCOMO II Linearen model	**Data Protector linearen model**
Faza načrtovanja in zahtev	Faza definicije zahtev
Faza načrtovanja produkta	Faza definicije specifikacij
	Faza implementacije produkta
Faza podrobnega načrtovanja produkta	Podrobno načrtovanje
Faza kodiranja in testiranja posameznih enot	Kodiranje in testiranje enot
Faza integracijskega testiranja in testiranja sistema	Integracijsko testiranje
	Del sistemskega testiranja
	Faza prevzema produkta
	Dokončano sistemsko testiranje
Faza namestitve produkta	Testiranje v okviru prevzema produkta
	Faza, v kateri se projekt zapre
Faza uporabe in vzdrževanja	Faza uporabe in vzdrževanja

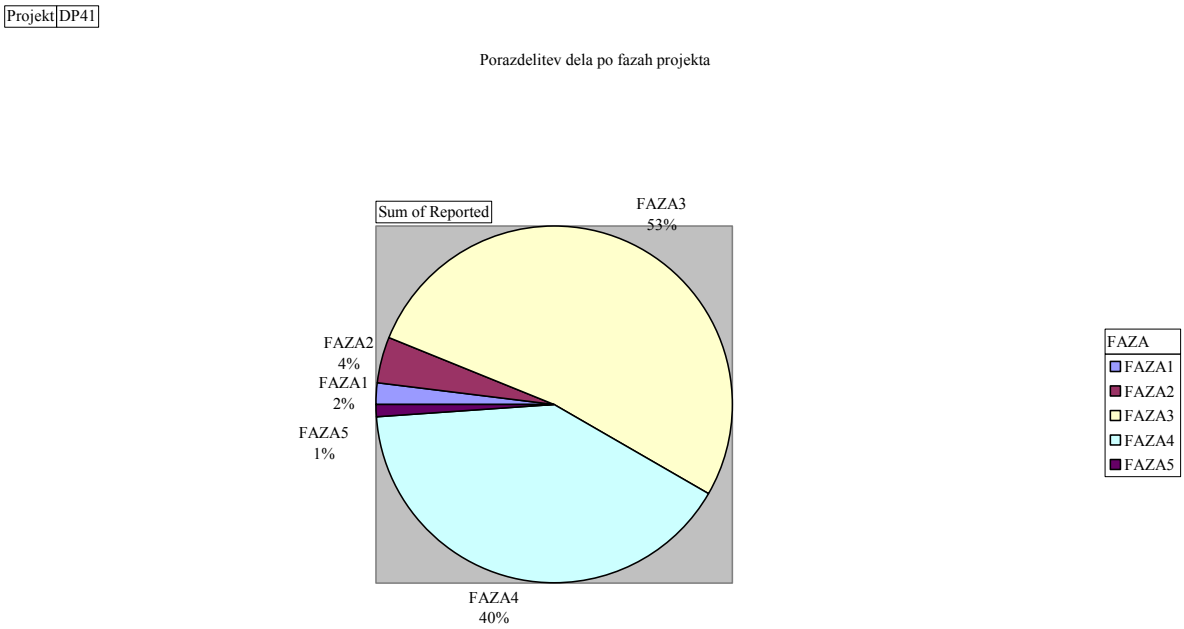
Tabela 23: Preslikava faz dveh linearnih modelov

4.4.4. Porazdelitev vloženega dela na projektu po fazah

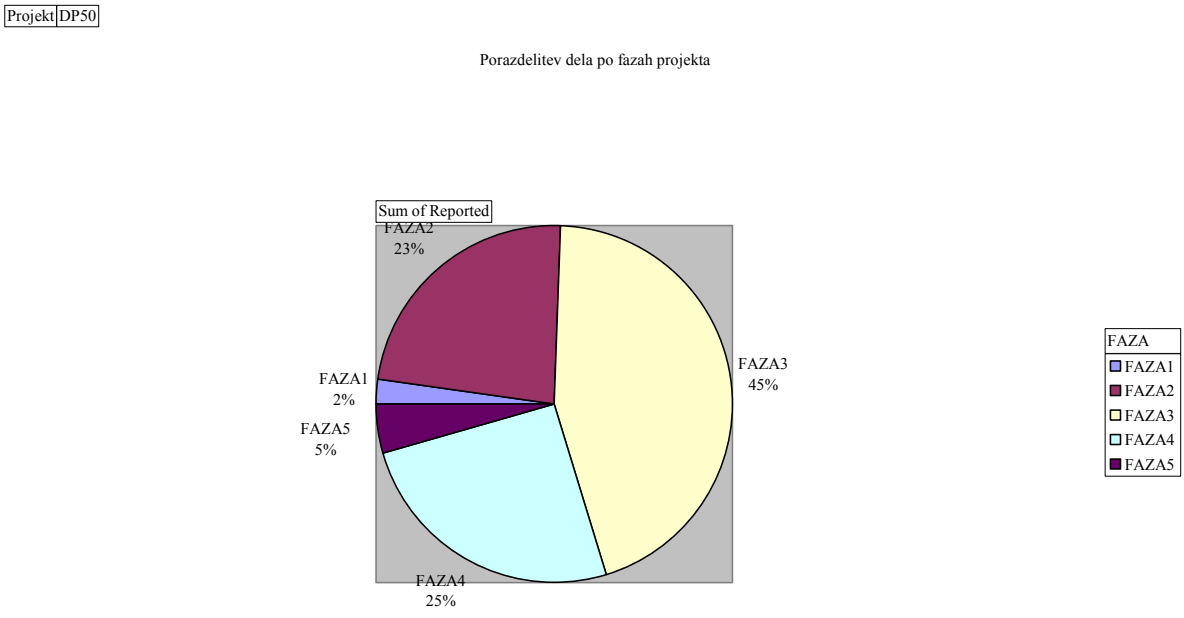
Zanimiva je tudi porazdelitev dela vloženega na projekt po posameznih fazah. Faza 1 in faza 2 sta praviloma kratki. Na večjih projektih lahko dosegajo nekje okrog 20 do 25 odstotkov dela vloženega v projekt. Na manjših projektih pa je ta odstotek praviloma manjši in se giblje okrog 10 odstotkov dela vloženega v projekt. Faza implementacije je praviloma najdaljša in obsega ponavadi okrog 50 odstotkov dela vloženega na projekt. Faza prevzema projekta obsega nekako do 30 odstotkov dela vloženega na projekt in faza, v kateri dokončno zapremo projekt načelom obsega le okrog 5 odstotkov dela na projektu.



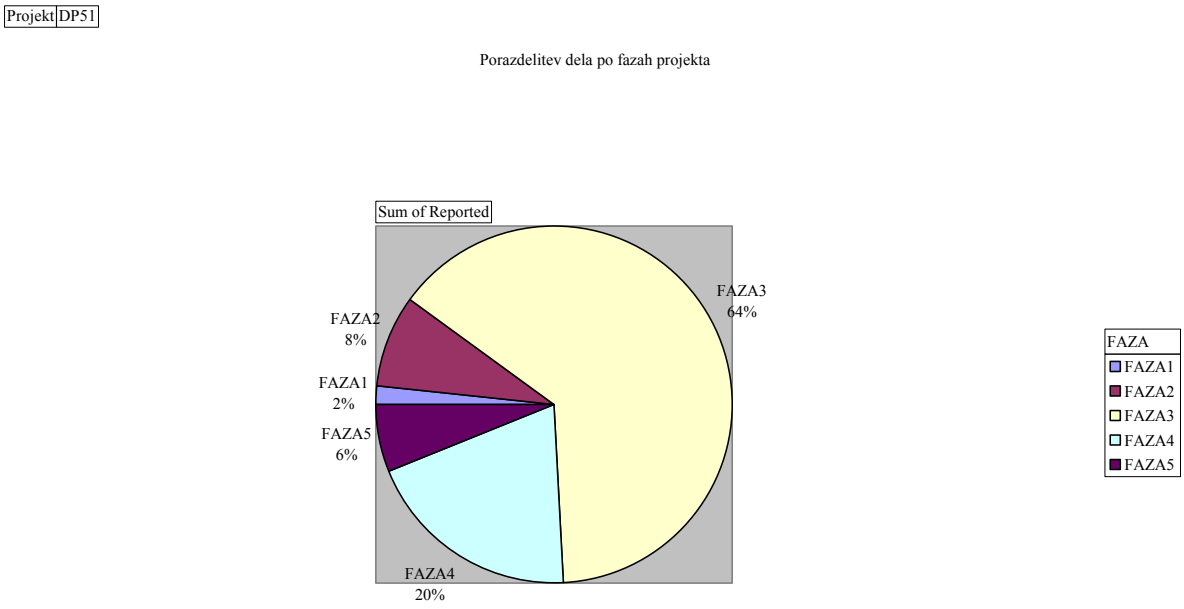
Slika 7: Porazdelitev dela na projektu DP40 po fazah



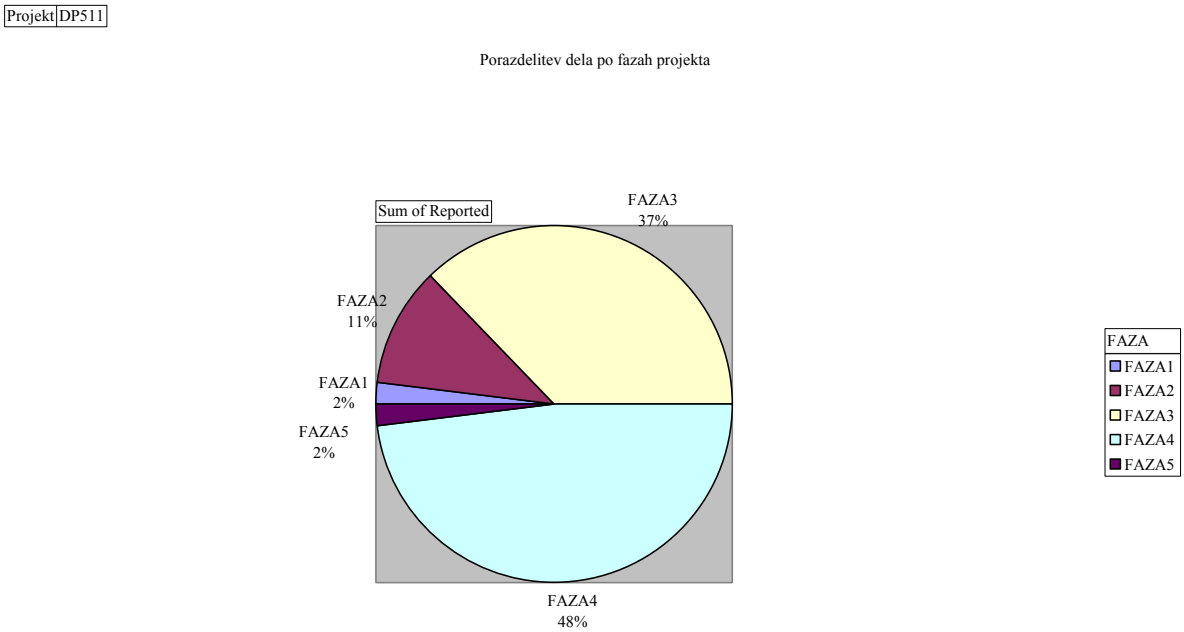
Slika 8: Porazdelitev dela na projektu DP41 po fazah



Slika 9: Porazdelitev dela na projektu DP50 po fazah



Slika 10: Porazdelitev dela na projektu DP51 po fazah



Slika 11: Porazdelitev dela na projektu DP511 po fazah

Takšni podatki o preteklih projektih so zelo koristni, ker lahko z njihovo pomočjo preverimo pravilnost ocen o potrebnem delu na projektu. Denimo, da pripravimo oceno novega produkta, za katerega predvidimo, da je potrebno le 30 odstotkov celotnega dela v fazi 3. V takšnem primeru, se moramo dejansko vprašati, ali je ta projekt res tako različen od prejšnjih in zato potrebujemo manj dela v fazi implementacije, ali pa smo mogoče naredili napačno oceno.

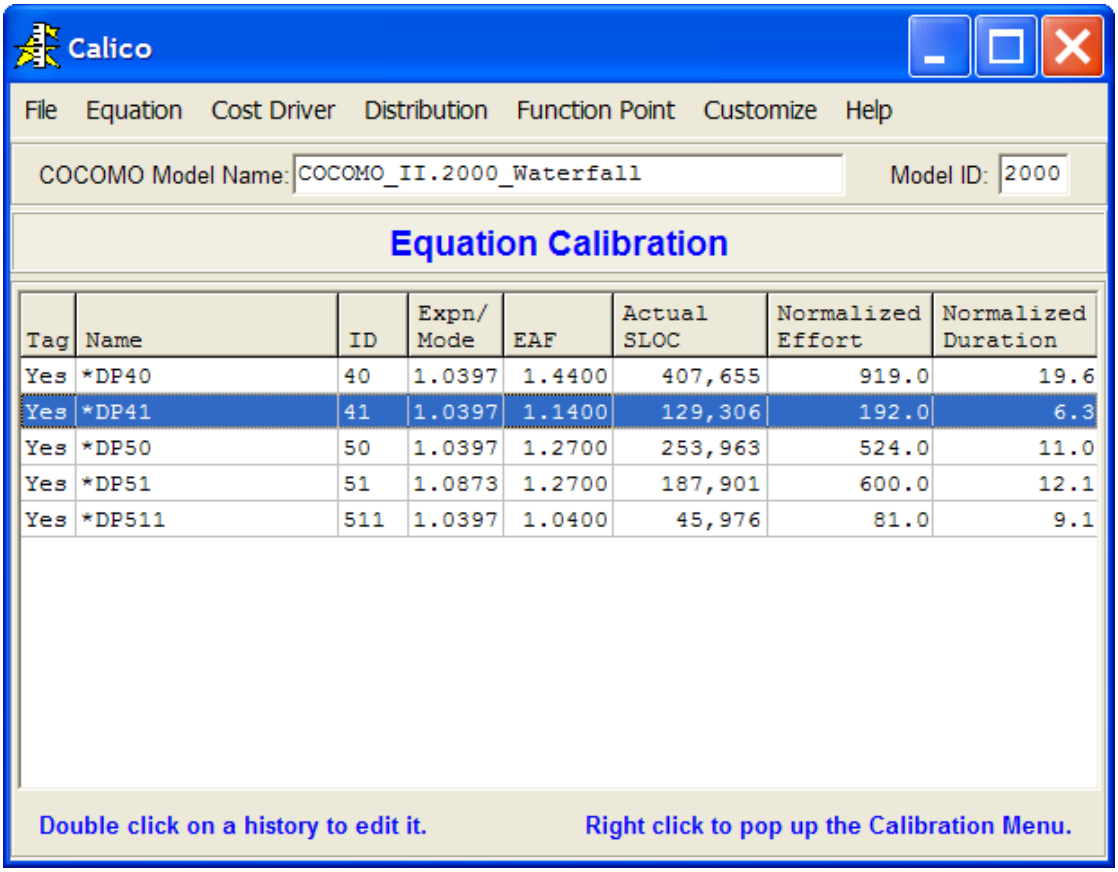
4.5. Kalibracija enačb za določitev ocene potrebnega dela in časa na projektu

Kalibriranje je eno izmed pomembnih korakov pri postavljanju parametrične ocene o ceni projekta. Omogoča nam, da lahko prilagodimo model lokalnim razmeram, za katere poskušamo definirati ocene. Kalibriranje vedno poteka na osnovi zgodovinskih podatkov o projektih. V primeru razvoja produkta Data Protector, sem zbrala podatke o zadnjih petih projektih:

- DP40 – Mambo projekt, ki je potekal od novembra 1999 do avgusta leta 2001. Mambo projekt je bil eden večjih projektov na razvoju programskega produkta Data Protector.
- DP41 – Tango projekt, ki je potekal od maja 2001 do decembra 2001. Šlo je za manjši produkt, cilj katerega je bil čim hitreje podpreti določene tehnologije.
- DP50 – Smart projekt, ki je potekal od septembra 2001 do septembra 2002. Spet je šlo za enega večjih projektov, ki pa se po svoji velikosti še vedno ne more meriti s projektom DP40.
- DP51 – Mt Blanc projekt, ki je potekal od aprila 2002, do junija 2003. Na začetku je bil tudi projekt DP51 mišljen kot kratek majhen projekt, vendar se je projekt zaradi nenehnih dodatnih zahtev zavlekel, določene dodatne zahteve, ki pa bi le preveč raztegnile projekt, so bile prenesene v naslednji projekt DP511.
- DP511 – IronEVA projekt, ki je potekal od decembra 2002 do septembra 2003. Spet je šlo za manjši projekt, pozne spremembe zahtev na projektu DP51 so bile preusmerjene v nov projekt.

Za osnovo sem uporabila model CALICO (CALIbrate COcomo). Osnova za kalibracijo so podatki o preteklih projektih. Podatki, ki so potrebni za to, da lahko kalibriramo model, so podatki o velikosti projekta, podatki o vloženem delu na projektu in pa podatkih o ocenjenih faktorjih dela in faktorjih eksponenta za pretekle projekte (Slika 12).

Velikost projekta je bila ocenjena glede na spremembe v kodi, ki so bile narejene pri posameznih projektih. Programski produkt je sestavljen iz posameznih modulov. Za vsakega od modulov sem na osnovi podatkov o spremembi modula, ocenila koliko vrstic v modulu je bilo spremenjenih, oziroma dodanih. S pomočjo dodanih in spremenjenih vrstic znotraj posameznega modula sem določila velikost posameznega modula v posameznem projektu. Upoštevala sem samo dejanske vrstice, ki sodijo k kodi.



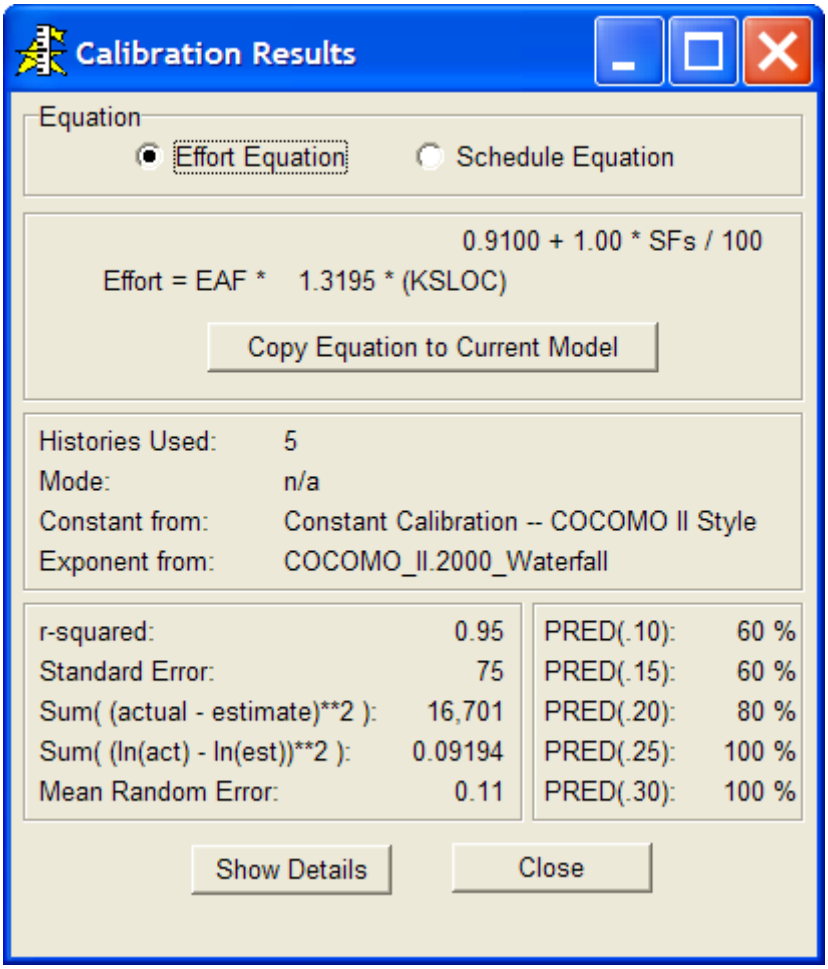
Slika 12: Osnova za kalibriranje Data Protector projektov

V začetku sem imela sicer v mislih drugačen pristop za ocenitev velikosti projekta. Nameravala sem upoštevati spremembe CM, DM in IM, kot jih opisuje model v

primeru ponovne uporabe kode. Ker pa je to zelo težka naloga v primeru, da stvari ocenjujemo za nazaj, sem se odločila, da upoštevam samo faktor spremembe kode, ne pa tudi spremembo načrta in integracije.

Za vsakega od projektov sem imela na voljo tudi podatek o porabljenem delu na projektu. Torej, koliko inženirskih mesecev je bilo porabljenih za izvedbo projekta.

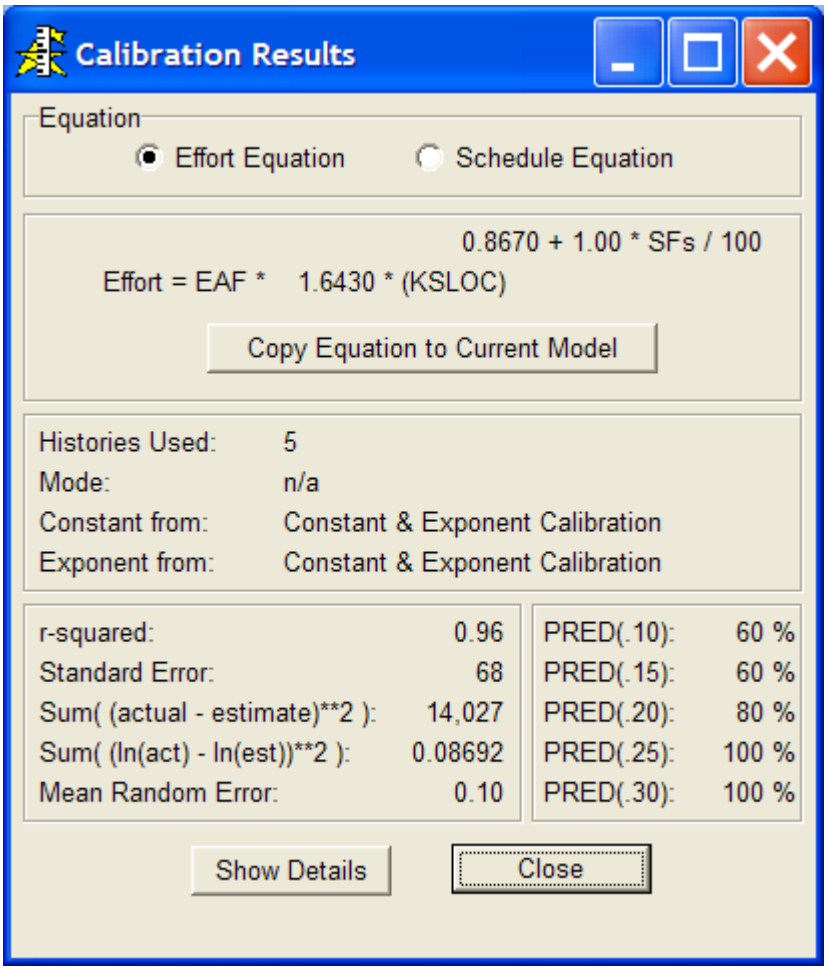
Za projekte sem določila tudi faktorje dela in faktorje eksponenta. Vendar pa sem se odločila za postavitev le-teh na vrednost, ki ni enaka povprečni, le v primerih, ko sem za to imela dejansko tehten razlog, saj sem ocenjevala le-te v času, ko so bili projekti že končani in nisem več imela popolne slike o njih.



Slika 13: Kalibriranje enačbe za potrebno delo (samo A)

Glede na to, da sem rekonstruirala velikosti projektov le za zadnjih pet projektov v verigi, sem prvotno nameravala kalibrirati le konstanto A. S pomočjo kalibracije je konstanta A prilagojena na vrednost 1.3195.

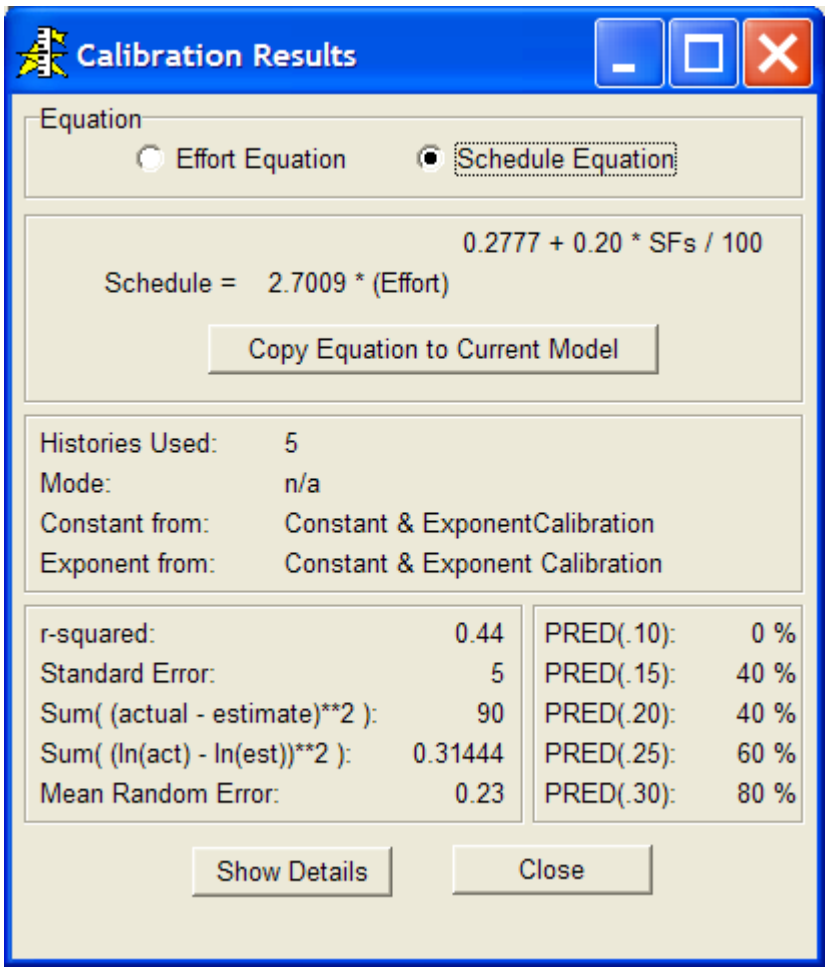
Kljub temu, da je priporočeno število projektov za kalibracijo obeh konstant A in B vsaj 7, sem se odločila, da poskusim tudi s kalibracijo obeh konstant modela za določanje potrebnega dela (Enačba 3-1). Razlog za to je predvsem, da so si projekti dejansko zelo podobni in da mogoče ravno zaradi te podobnosti ne potrebujemo tolikšnega števila podatkov. Slika 14 prikazuje rezultat kalibracije obeh konstant. Vrednost konstante A je bila kalibrirana na 1.6430, vrednost konstante B pa na 0.8670. Vidimo, da je drugi model za spoznanje boljši. Natančnost obeh modelov je zadovoljiva. Statistika PRED (x/100) nam določa odstotek projektov, za katere ocena ne odstopa od x% dejanske vrednosti. Večinoma se za primerjavo vzame vrednost PRED (0.25). Torej koliko projektov ne bo odstopalo v oceni za več kot 25%. Vidimo da ima naš model vrednost PRED (0.25) 100%. COCOMO model zahteva, da je PRED (0.25) večji od 63%.



Slika 14: Kalibriranje enačbe za potrebno delo

Nadaljevala sem tudi s kalibracijo enačbe za določanje časa potrebnega za razvoj, vendar se napake modela za določanje časa večje, kot pri oceni potrebnega dela V

tem primeru kalibriramo konstanto C in D modela za določanje časa potrebnega za razvoj (Enačba 3-2). Vrednost faktorja C je v modelu COCOMO II postavljena na 3.67, postopek kalibracije za projekte Data Protector pa jo je postavil na vrednost 1.8285. Vrednost konstante D je 0.28 za model COCOMO II in 0.2778 za Data Protector model. Vidimo, da je napoved modela v okviru 25% napake le za 60% projektov. To je nekaj manj kot zahteva model COCOMO II, je pa še vedno relativno dobra ocena.



Slika 15: Kalibriranje enačbe za terminski plan

4.6. Kalibriran COCOMO II parametrični modela za Data Protector projekte

S pomočjo zgoraj opisanega postopka za kalibracijo lahko sedaj definiramo model za parametrično oceno projekta Aragon.

$$PM_{NS} = 1.6430 \times Size^{E} \times \prod_{i=1}^{n} EM_i$$ **Enačba 4-1**

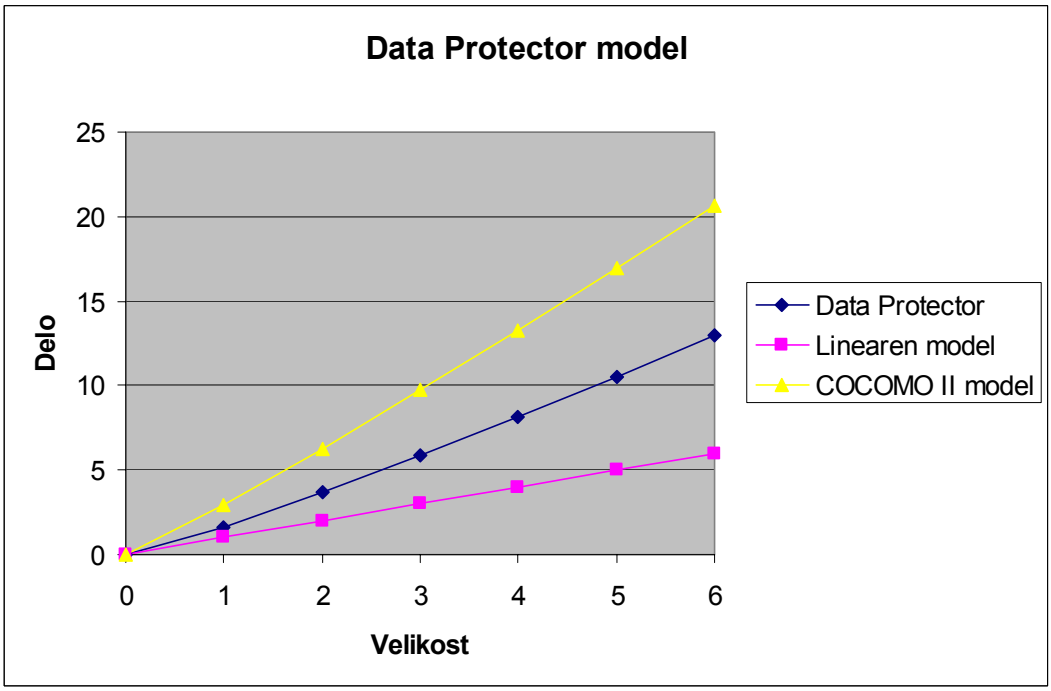
kjer je $E = 0.8670 + 0.01 \times \sum_{j=1}^{5} SF_j$

Če upoštevamo še oceno faktorjev dela in faktorjev eksponenta, potem lahko model opišemo z naslednjo enačbo:

$$PM_{NS} = 1.6430 \times Size^{E}$$ **Enačba 4-2**

$$E = 0.867 + 0.01 \times 17.73 = 1.154$$

Slika 3 nam prikazuje primerjavo med 3 modeli za oceno potrebnega dela: model, ki je bil kalibriran za Data Protector projekte, linearen model in model COCOMO II, v primeru, da ne bi opravili postopka kalibracije.



Slika 16: Primerjava modelov

S pomočjo kalibracije pridemo do naslednjega modela za določanje trajanja projekta:

$$TDEV_{NS} = 1.8285 \times (PM_{NS})^{F}$$ **Enačba 4-3**

$$F = 0.2778 + 0.2 \times 0.01 \times \sum_{j=1}^{5} SF_j = 0.2778 + 0.2 \times (1.154 - 0.9670) = 0.3152$$

Če upoštevamo še faktor SCED, potem je enačba za določanje trajanje projekta naslednja:

$$TDEV_{NS} = 1.8285 \times (PM_{NS})^{0.3152} \times \frac{SCED\%}{100}$$ **Enačba 4-4**

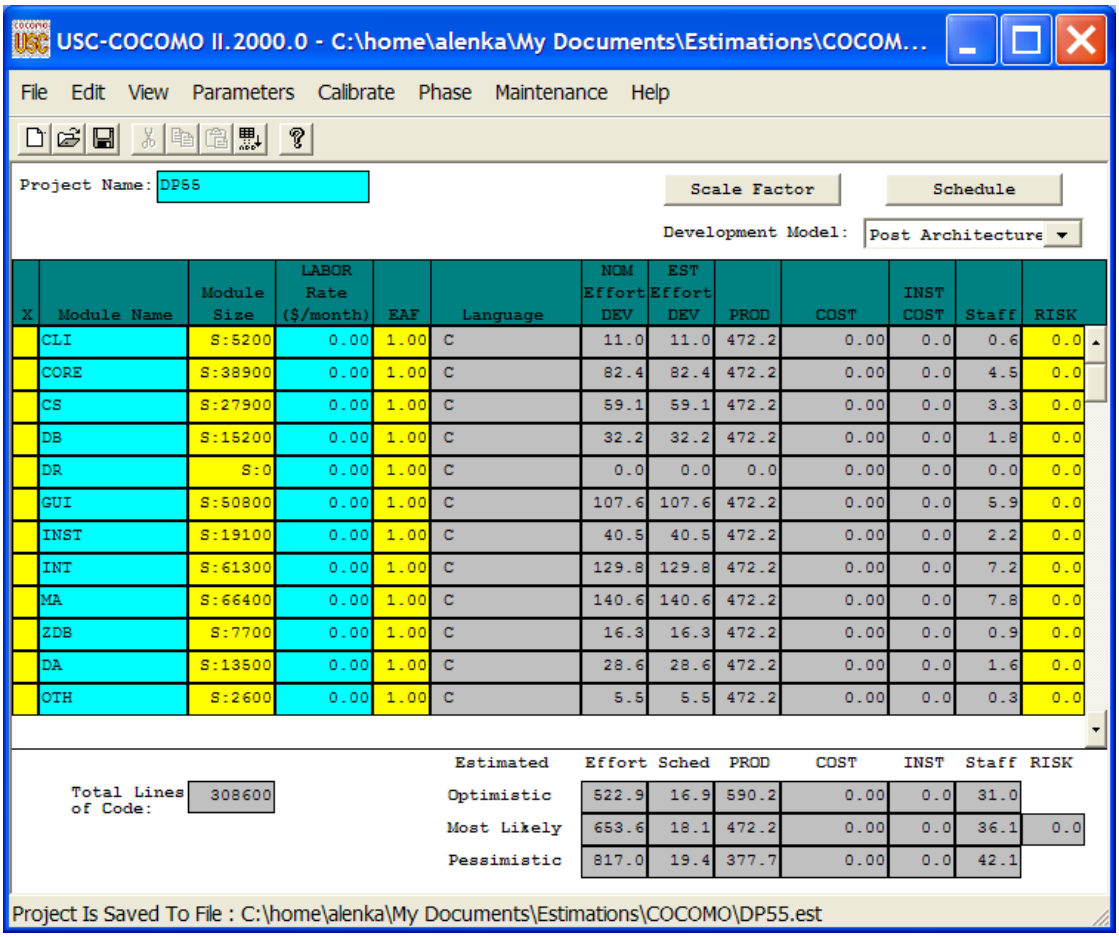
4.7. Ocena dela na projektu Aragon

Sedaj imamo na voljo zadosti podatkov, da lahko naredimo dejansko parametrično oceno s pomočjo kalibriranega parametričnega modela COCOMO II. Ocena je bila narejena s pomočjo predhodno določene velikosti projekta, ki je bila ocenjena v razdelku 4.1, na osnovi definiranih faktorjev dela in faktorjev eksponenta in kalibriranega modela COCOMO II za projekte Data Protector. Za velikost projekta je bila vzeta ocena ekvivalenta vrstic kode ($KSLOC_{EQ\_B}$), v primeru, da ne upoštevamo faktorjev SU in UNFM. Razlog za to je, da je bil Data Protector model kalibriran na osnovi dejansko spremenjenih vrstic kode. V razdelku 4.1.7 smo že opisali razmerja med ekvivalentom vrstic kode in dejansko spremenjenimi vrsticami kode.

Slika 17 prikazuje oceno potrebnega dela za projekta Aragon z modelom COCOMOII. Na sliki vidimo, da je delo, potrebno za dokončanje produkta Aragon, ocenjeno na 653 EM, v primeru pesimistične ocene na 817 EM in v primeru optimistične napovedi na 522 EM. Ker je bila ocena faktorjev dela povprečna (glej razdelek 4.3.4.4), vidimo da je nominalno delo enako ocenjenemu delu na posameznih modulih.

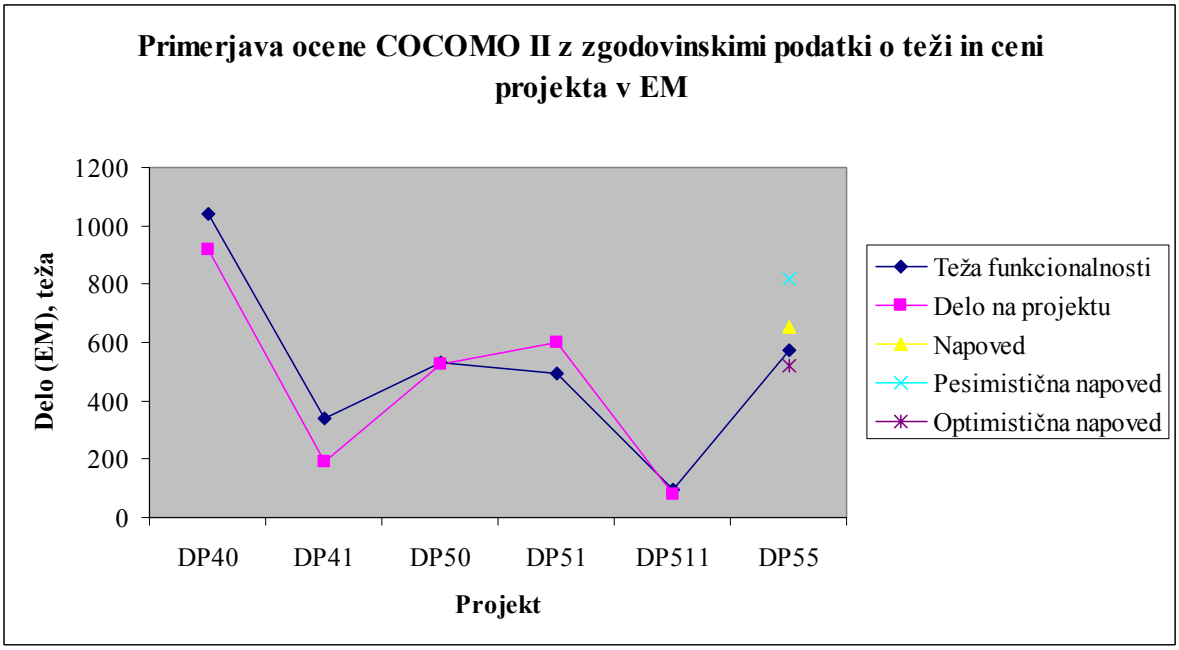
Ocenimo lahko, da je ocena dokaj realna in v skladu s pričakovanim. Pogledamo lahko tudi na graf, ki kaže primerjavo med težo projekta, kot je bila opisana v 4.1.1 in dejanskim vloženim delom na projektu. Če vnesemo napoved, ki je bila narejena s pomočjo modela COCOMO II (glej Slika 18) vidimo, da je dobra napoved in da lahko z

veliko gotovostjo rečemo, da bo projekt za uspešen konec zahteval količino dela, ki je bilo ocenjena.



Slika 17: Ocena projekta Aragon s pomočjo kalibriranega modela COCOMO II

Prav tako je tudi ocena trajanja projekta dosti realno postavljena. Seveda, če upoštevamo še faktor SCED, ker se bo trajanje projekta Aragon zaradi zmanjševanja inženirjev podaljšalo. Faktor SCED sem za projekt Aragon ocenila na visoko vrednost, kar ne prinaša dodatnega dela v oceni, poveča pa potreben čas za razvoj za 30%.



Slika 18: Primerjava ocene COCOMO II modela z zgodovinskimi podatki o teži in ceni projekta v EM

4.8. Zaključek

V tem poglavju je bil opisan način prilagoditve COCOMO II parametričnega modela za projekte na razvoju programskega produkta Data Protector. Zaključimo lahko, da nam zgodovinski podatki, ki smo jih uspeli zbrati na teh projektih in pa tudi podatki, ki jih je bilo moč rekonstruirati v času nastajanja naloge, dejansko veliko povedo in sporočajo. Ta primer lepo kaže smisel teh podatkov in tudi vrednost, ki jo ti podatki imajo. Če znamo na podatke pogledati s pravega zornega kota, če znamo oceniti, kaj je pomembno pri določanju končnega rezultata, si lahko konkretno pri ocenjevanju potrebnega vloženega dela v projekt ogromno pomagamo in na dokaj hiter način pridemo do ocene, ki nam pomaga pri določanju končne cene.

S pomočjo parametričnega modela lahko projektni vodja pride do ocene dokaj samostojno in pri pogajanjih o ceni posameznega dela na projektu hitro izloči dele, ki zahtevajo bistveno več, oziroma manj časa od pričakovanega. V takšnem primeru seveda ne gre parametričnemu modelu slepo zaupati in je potrebno poslušati argumente za in proti. Je pa naloga projektnega vodje v tem smislu lahko nekako razbremenjena, saj so postavljeni okviri pričakovanega.

Tema je vsekakor zanimiva in aktualna. Trenutni rezultati dela pa predstavljajo le en korak v bolj organiziran način določanje ocene potrebnega dela na projektu.

5. Priloga: Struktura arhiva Data Protector

Za to, da sem lahko zbrala zadosti podatkov o projektih, ki sem jih obdelovala, so bili pomembni prav zgodovinski podatki. Vsi podatki niso bili neposredno na voljo, ampak jih je bilo prej potrebno zbrati in primerno predstaviti, da sem jih lahko uporabila kot osnovo za ocenjevanje potrebnega dela.

Predvsem je bilo potrebno zbrati naslednje podatke:

- Podatke o projektni ekipi, ki nam pomagajo pri definiciji faktorjev osebja.
- Podatki o kontrolnih točkah na projektu, ki nam omogočajo sledenju terminskega plana. Ocenimo lahko tudi, kako dobro smo projekt planirali na samem začetku.
- Podatki o delu, ki je bilo opravljeno na projektu po fazah. Ti podatki nam omogočajo ocenjevati delež dela na projektu po fazah na projektu.
- Podatki o obsegu projekta, ki nam omogočajo določiti velikost projekta.

- Podatki o podprtih platformah produkta, ki nam omogočajo primerjavo med produkti glede na podrte platforme. Ta podatek uporabimo tudi pri ocenjevanju kompleksnosti novega produkta, kot eno izmed komponent.
- Podatki o testiranju produkta.
- Podatki o problemih, ki so bili najdeni na projektu v okviru testiranja. Ti nam omogočajo predvsem primerjavo med projekti z vidiki kvalitete testnih verzij projekta.
- Podatki o pregledih produkta s strani ekipe za vzdrževanje. Ti so zopet pomembni za oceno samega produkta s stališča kvalitete.

Prvi so podatki o številu spremenjenih vrstic na posameznem projektu po modulih. Glede na to, da vsak naslednji produkt prevzame kodo prejšnjega, je bilo potrebno predvsem oceniti razliko na kodi med dvema projektoma. Pomembna je tudi tabela dodanih funkcionalnosti na projektu, ki nam je ravno tako služila kot pripomoček pri delu, podatki o kontrolnih točkah na projektu in podatki o vloženem delu na projektu. Na osnovi teh dveh vrst podatkov, sem lahko ocenila velikost projekta in pa tudi vloženo delo v projekt.

Ob koncu projektov, se zbira tudi serija različnih podatkov, ki sem jih tudi opisala v tem poglavju, tako da imamo ne enem mestu zbrane vse informacije, ki so nam na voljo za projekte Data Protector.

5.1. Definicija in opis strukture arhiva

5.1.1. Splošne informacije

Formular 1.0: Splošne informacije o projektu Ime projekta: ID projekta:	
Ime projekta	Ime projekta, ki se uporablja med samim razvojem
Id	Identifikacijska številka projekta
Ime produkta	Ime končnega produkta
Verzija produkta	Verzija produkta
Podjetje	Podjetje, kjer se je projekt razvijal
Poslovno področje	Poslovno področje, kjer se je projekt razvijal
Projektni vodja	Oseba, ki je vodila projekt
Stranka	Stranka, ki je naročila projekt
Kontaktna oseba	Oseba, ki je skrbela za komunikacijo pri naročniku
Tip projekta	Nov projekt, nova verzija projekta, vzdrževanje in popravki ...
Način razvoja	Življenjski cikel, ki je bil izbran za vodenje projekta
Kdaj v procesu so se zbirali podatki	Med projektom, ob koncu faz projekta, na koncu projekta, kasneje...
Cilji projekta	Opiši glavne cilje projekta
Reference	Vpiši reference na dokumente o projektu
Arhivar	Oseba, ki je zbirala podatke o projektu
Datum	Datum, ko so bili podatki zbrani

Tabela 24: Splošne informacije o projektu

5.1.2. Projektna ekipa

Projektna ekipa za razvoj produkta Data Protector je sestavljena iz več skupin. Vsaka izmed skupin skrbi za eno področje pri razvoju programske opreme. Razlikujemo naslednje skupine:

- Skupino razvijalcev (DEV - Development) .
- Skupino za zagotavljanje kvalitete (QA – Quality Assurance).
- Skupino zadolženo za dokumentacijo (LP – Learning Products).

- Skupino zadolženo za upravljanje konfiguracije (CM - Configuration Management).
- Skupina zadolžena za vodenje projekta in ekip (PM – Project Management in TM – Team Management).

Tabela 25 vsebuje informacijo o inženirjih, ki so delali na projektu v kateri od prej naštetih skupin.

Formular 2.0 : Struktura projektne ekipe Ime projekta: ID projekta:	
Ekipa	Število ljudi
MGMT	
DEV	
QA	
LP	
CM	

Tabela 25: Struktura projektne ekipe

Tabela 26 vsebuje podatke o posebnih zadolžitvah in izkušenosti posameznikov v projektni ekipi:

- Vloga: Vodilni inženir (LE), Inženir (E), Inženir Začetnik (AE), Arhitekt (A), Projektni vodja (PM), Vodja ekipe (TM), Vodja Pregledov kode (CRL).
- Skupina: Razvoj (DEV), Dokumentacija (LP), Testiranje (QA), Upravljanje konfiguracije (CM), Vodenje (MGMT).
- Zadolžitev: V primeru posebnih zadolžitev – lastnik funkcionalnosti (FO).
- Delovne izkušnje: Število let delovnih izkušenj.
- Poznavanje produkta: Število let dela na projektu.
- Poznavanje orodij: Število let dela s podobnimi orodji.
- Poznavanje platforme: Izkušnje pri delu na podobni platformi.

Formular 2.1: Projektna ekipa (II) Ime projekta: ID projekta:							
Ime Priimek	Vloga	Skupina	Zadolžitev	Delovne izkušnje	Poznavanje produkta	Poznavanje orodij	Poznavanje platforme

Tabela 26: Zadolžitve in izkušenost projektne ekipe

5.1.3. Projekt po fazah

1. Trajanje projekta po fazah

Razvoj poteka po klasičnem načinu razvoja programskih projektov (linearni model). Projekt je razdeljen na več faz. Faze projekta si sledijo v naslednjem vrstnem redu:

1. Faza 1 - Definicija zahtev (REQ – Project Requirements Phase).
2. Faza 2 - Specifikacija (SPEC – Project Specification Phase).
3. Faza 3 - Implementacija (IMPL – Project Implementation Phase).
4. Faza 4 - Presoja ustreznosti (ACCEP – Project Acceptance Phase).
5. Faza 5 - Zapiranje projekta (CLOSE – Project Closing Phase).
6. Fazi zapiranja projekta sledi vzdrževanje projekta (MAINT – Project Maintenance Phase).

Pri mejnikih projekta so pomembni naslednji podatki:

- Planirani mejniki na začetku projekta, takrat ko začnemo s projektom. V tem trenutku ponavadi vemo, kaj od naslednjega projekta pričakujemo, katere so dejansko pomembne funkcionalnosti, ki jih želimo v tem času narediti in pa tudi kdaj želimo z naslednjo verzijo produkta priti na trg.
- Mejniki, ki so sprejeti bo koncu faze 2. Ti so dejansko najbolj pomembni, saj ob koncu faze 2 na projektu ponavadi določimo končni datum projekta in

damo tudi zagotovila, da bo projekt narejen v predvidenem roku, če stranka ne bo spreminjala v preveliki meri svojih zahtev.

- Mejniki, do katerih je bila dejansko določena faza končana.

Formular 3.0 : Mejniki projekta Ime projekta: ID projekta:			
	Planirani mejniki (konec faze 1)	Planirani mejniki (konce faze 2)	Dejanski mejniki
Začetek projekta			
Konec faze 1 REQ			
Konec faze 2 SPEC			
Konec faze 3 IMPL			
Konec faze 4 ACCEPT			
Konec faze 5 CLOSE			
Konec faze 6 MAINT			
Zamrznitev funkcionalnosti			
Javna beta verzija			
Proizvodna verzija MR			
Produkt na trg SR			

Tabela 27: Mejniki projekta

2. Delo opravljeno na projektu po skupinah in fazah

Tabela 28 nam prikazuje informacijo, kako je bilo delo porabljeno po fazah znotraj posameznih skupin.

Formular 3.1 : Delo na projektu po skupinah in fazah Ime projekta: ID projekta:						
	Delo v DEV ekipi	Delo v CM ekipi	Delo v QA ekipi	Delo v LP ekipi	Delo v MGMT ekipi	Skupaj
REQ						
SPEC						
IMPL						
ACCEPT						
CLOSE						
MAINT						
Skupaj						

Tabela 28: Delo na projektu po skupinah in fazah

5.1.4. Obseg projekta

Na vsakem projektu se razvija seznam novih funkcionalnosti. Za vsako od funkcionalnosti so pomembni naslednji podatki:

- Lastnik funkcionalnosti, ki je odgovoren za popolno izpeljavo nove funkcionalnosti znotraj projekta.
- Skupina, ki ji lastnik funkcionalnosti pripada.
- Nivo spremembe, ki je potrebna, da dodamo funkcionalnost v produkt.
- Vpliv funkcionalnosti na posamezne module, ter modul, ki nosi največjo težo sprememb.

Formular 4.0 : Obseg projekta Ime projekta: ID projekta:															
ID	Funkcionalnost	Skupina	Lastnik	Nivo spremembe	CORE	DB	CS	MA	DA	GUI	CLI	INST	INT	ZDB	OTH

Tabela 29: Obseg projekta

Pomembno je tudi spremljanje posameznih funkcionalnosti v smislu, ali je funkcionalnost pripravljena na integracijski testiranje. Beležimo naslednje podatke:

- Ime funkcionalnosti.
- Lastnik funkcionalnosti.
- Prvi planirani datum, ko je funkcionalnost pripravljena za integracijsko testiranje.
- Prestavljeni planirani datum (Novi FF – Functionality Freeze datum).
- Datum, je bila funkcionalnost predstavljena vodstveni ekipi na projektu (Končni FF datum).
- V primeru, da testna ekipa ni zadovoljna z izdelano funkcionalnostjo, jo lahko vrne lastniku funkcionalnosti in prekine testiranje, dokler se pomanjkljivosti ne odpravijo (FCL – Functionality Checklist zavrnjena).
- Ko se pomanjkljivosti odpravijo, se funkcionalnost zopet vrne v testiranje (FCL vrnjen).
- V zadnji koloni izračunamo zaostanek v tednih.

Formular 4.1 : Spremljanje funkcionalnosti

Ime projekta:

ID projekta:

Funkcionalnost	Lastnik	Planirani FF datum	Novi FF datum	Končni FF datum	Zaostanek (tedni)	FCL zavrnjen	FCL vrnjen	Zaostanek (tedni)

Tabela 30: Spremljanje funkcionalnosti

Formular 4.2 : Spremembe na kodi po modulih Ime projekta: ID projekta:													
	CLI	CORE	CS	DA	DB	DR	GUI	INST	INT	MA	ZDB	OTH	Skupaj
Velikost pred projektom (KSLOC)													
Delta (KSLOC)													
Velikost po projektu (KSLOC)													
Sprememba kode (%)													

Tabela 31: Spremembe na kodi po modulih

Pomemben podatek za razumevanje obsega projekta so tudi podatki o spremenjenih vrsticah kode (Tabela 31).

Med samim postopkom ocenjevanja je pomembna tudi tabela, s pomočjo katere ocenimo predvideno velikost projekta. Postopek, kako pridemo do ocene, je opisan v razdelku 4.1, sama tabela pa je tudi shranjena kot del arhiva projekta.

Formular 4.3 : Ocena velikosti projekta Ime projekta: ID projekta:													
	SKUPAJ	CLI	CORE	CS	DA	DB	DR	GUI	INST	INT	MA	OTH	ZDB
KSLOC													
DM													
CM													
IM													
SU													
UNFM													
AAF													
AAM_E													
AAM_B													
$KSLOC_{EQ\_E}$													
$KSLOC_{EQ\_B}$													

Tabela 32: Ocena velikosti projekta

5.1.5. Platforme produkta

Ker je projekt Data Protector na trgu že deset let, je zanimiv tudi podatek, kako raste njegova kompleksnost z vidika podprtih platform, operacijskih sistemov, na katerih tečejo različni moduli produkta Data Protector.

Formular 5.0 : Ocena kompleksnosti produkta glede na podprte platforme Ime projekta: ID projekta:								
	Začetno število podprtih OS	Število dodanih OS	Število odvzetih OS	Končna število podprtih OS	Teža	Dodana kompleksnost	Odvzeta kompleksnost	Končna kompleksnost
CORE								
Manager								
MoM								
IS								
GUI								
CLI								
DA								
MA								
SAP R/3								
Oracle								
Exchange								
SQL								
Sybase								
Informix								
DB2								
SAPDB								
Lotus								
SMB								
Symmetrix								
Sure Store								
VA								
XP								

Tabela 33: Kompleksnost produkta glede na podprte platforme

5.1.6. Testiranje produkta

1. Faze testiranja

Testiranje na projektu poteka v več fazah.

1. Integracijsko testiranje (INTEG), kjer se preveri ustreznost vmesnikov med različnimi moduli.
2. Sistemsko testiranje (SYSTEM), kjer skušamo preveriti ustreznost celotnega sistema.
3. Beta sistemsko testiranje – testiranje pri strankah.
4. Testiranje v okviru prevzema produkta (ACCEPT).

Tabela 34 vsebuje podatke o planiranih, izvršenih in uspešnih testih znotraj posamezne testne faze.

Formular 6.0 : Število izvršenih testov po fazah Ime projekta: ID projekta:			
	Število planiranih testov	Število izvršenih testov	Število uspešnih testov
INTEG			
SYSTEM			
BETA SYSTEM			
ACCEPT			

Tabela 34: Število izvršenih testov po fazah

2. Opis strukture testov

Formular 6.1 : Opis strukture testov Ime projekta: ID projekta:	
Dokumentirani testni scenariji	
Novi testni scenariji	
Izvršeni testi v integracijski fazi	
Izvršeni testi v sistemski fazi	
Skupaj zabeležene izvršitve testov	
Skupaj zabeležene izvršitve testov (brez ponovitev)	
Avtomatski testi	
Novi avtomatski testi	
Število testnih skript (Perl)	
Število vrstic Perl skript	
Število Visual skript	
Število vrstic Visual skript	

Tabela 35: Opis strukture testov

5.1.7. Statistika problemov v DDTS

Tabela 36 vsebuje opis prijavljenih problemov na projektu s strani testne ekipe. Problemi so ločeni glede na inženirja, ki ga je prijavil in glede na resnost napake, ki se ocenjuje z vidika uporabnika.

Formular 7.0: Prijavljeni problemi Ime projekta: ID projekta:					
Prijavil	CRITICAL	SERIOUS	MEDIUM	LOW	Skupaj
…					
Skupaj					

Tabela 36: Prijavljeni problemi

Tabela 37 prikazuje probleme, ki so bili preverjeni. Problemi so zopet grupirani po inženirju, ki je rešen problem še enkrat preveril in po kritičnosti problemov.

Formular 7.1: Verificirani problemi Ime projekta: ID projekta:					
Preveril	CRITICAL	SERIOUS	MEDIUM	LOW	Skupaj
…					
Skupaj					

Tabela 37: Preverjeni problemi

Tabela 38 opisuje stanje baze problemov ob koncu projekta. V tabeli je več DDTS projektov, vsi pa so namenjeni enemu dejanskemu projektu.

Formular 7.2: Stanje DDTS baze ob koncu projekta Ime projekta: ID projekta:										
DDTS Projekt	A	C	D	N	O	P	R	V	Z	Skupaj
Skupaj										

Tabela 38: Stanje DDTS baze ob koncu projekta

5.1.8. Pregled novega produkta s strani inženirjev, ki delajo na vzdrževanju produkta

Tabela 39 prikazuje končno stanje problemov, ki so bili najdeni s strani ekipe, ki dela na vzdrževanju v okviru pregleda produkta, glede na resnost napake .

Formular 8.0: Najdeni problemi po teži problema Ime projekta: ID projekta:					
Teža napake	A	C	D	V	Skupaj
LOW					
MEDIUM					
SERIOUS					
CRITICAL					
Skupaj					

Tabela 39: Skupina za vzdrževanje, najdeni problemi po teži problema

Tabela 40 prikazuje končno stanje problemov, ki so bili najdeni s strani ekipe, ki dela na vzdrževanju, v okviru pregleda produkta, glede na modul, ki je bil vzrok napake .

Formular 8.1: Najdeni problemi po modulih Ime projekta: ID projekta:					
Modul	A	C	D	V	Skupaj
…					
Skupaj					

Tabela 40: Skupina za vzdrževanje, najdeni problemi po modulih

5.1.9. Kvaliteta produkta na trgu

Tabela 41 nam prikazuje probleme, ki so jih stranke prijavile po prvih šestih mesecih na trgu. Predvsem nas zanimajo resni in kritični problemi.

Formular 9.0: Najdeni problemi v prvih šestih mesecih produkta na trgu po teži problema Ime projekta: ID projekta:					
Teža napake	A	C	D	V	Skupaj
LOW					
MEDIUM					
SERIOUS					
CRITICAL					
Skupaj					

Tabela 41: Problemi na produktu v prvih šestih mesecih po teži problema

Tabela 42 nam prikazuje probleme, ki so jih stranke prijavile po prvih šestih mesecih na trgu po modulih, v katerih je bil vzrok za napako.

Formular 9.1: Najdeni problemi v prvih šestih mesecih produkta na trgu po modulih Ime projekta: ID projekta:					
Modul	A	C	D	V	Skupaj
…					
Skupaj					

Tabela 42: Problemi na produktu v prvih šestih mesecih po modulih

5.1.10. Upravljanje konfiguracije

Formular 10.0 : Upravljanje konfiguracije Ime projekta: ID projekta:	
Število zahtev po zlivanju kode	
Število zahtev po prevajanju kode	
Število dobrih »buildov« HPUX	
Število dobrih »buildov« Solaris	
Število dobrih »buildov« NT	
Število uporabnih »buildov« HPUX	
Število uporabnih »buildov« Solaris	
Število uporabnih »buildov« NT	
Število neuporabnih »buildov« HPUX	
Število neuporabnih »buildov« Solaris	
Število neuporabnih »buildov« NT	

Tabela 43: Upravljanje konfiguracije

5.1.11. Opis končne strukture produkta

Formular 11.0 : Programi, paketi in platforme Ime projekta: ID projekta:	
Velikost paketa NT	
Velikost paketa UX	
Velikost paketa Solaris	
Število izvršljivih datotek v produktu NT	
Število izvršljivih datotek v produktu UX	
Število izvršljivih datotek v produktu UX	
Število izvršljivih datotek v produktu NT	
Število paketov za namestitev (»push« instalacija)	
UNIX platforme za prevajanje	
Windows platforme za prevajanje	
MPE platforme za prevajanje	
Novel platforme za prevajanje	

Tabela 44: Programi, paketi in platforme

Formular 11.1 : Projektna dokumentacija Ime projekta: ID projekta:		
	Število dokumentov	Število strani
Število novih raziskav		
Zahteve (FURPS)		
Eksterna specifikacija (ERS)		
Načrti (HLD)		
Načrti (LLD)		
Interna dokumentacija (IRS)		

Tabela 45: Projektna dokumentacija

Formular 11.2 : Tiskana dokumentacija Ime projekta: ID projekta:	
Število priročnikov	
Število strani	
Velikost PDF	
Velikost PS	

Tabela 46: Tiskana dokumentacija

Formular 11.3 : Pomoč uporabnikom v grafičnem vmesniku Ime projekta: ID projekta:	
Število datotek	
Število obravnavanih tem	
Število spremenjenih tem	
Število novih tem	
Število odstranjenih datotek	
Število besed	

Tabela 47: Pomoč uporabnikom v uporabniškem vmesniku

Formular 11.4 : Navodila za uporabo CLI Ime projekta: ID projekta:	
Število navodil	
Število novih navodil	
Število spremenjenih navodil	
Število odstranjenih navodil	
Število besed	

Tabela 48: Navodila za uporabo CLI

Formular 11.5 : Lokalizacija Ime projekta: ID projekta:	
Podprti jeziki	
Podprte platforme	

Tabela 49: Lokalizacija

6. Uporabljene okrajšave

AA	Percentage of reuse effort due to assessment and assimilation
AAM	Adaptation Adjustment Modifier
ACAP	Analyst Capability
ACCEPT	Project Acceptance Phase
AEXP	Application Experience
ASLOC	Adaptec Source Lines of Code
CALICO	CALIbrate COcomo
CASE	Computer Added Software Engineering
CDR	Critical Design Review
CLOSE	Project Close Phase
CM	Percentage of Code Modified
CD	Cost Driver

CM	Configuration Management
CMM	Capability Maturity Model
COCOMO	COnstructive COst Model
COTS	Commercial Of The Shelf
CPE	Customer Product Experience
DATA	Database Size
DDTS	Distributed Defect Tracking System
DEV	Development
DM	Percentage of Design Modified
DOCU	Documentation match to life cycle need
EM	Engineer Month
EM_i	Effort Multiplier
ERS	External Reference Specification
ESLOC	Equivalent Source Lines of Code
FCL	Functionality Check List
FF	Functionality Freeze
FP	Function Point
FPA	Function Point Analysis
FURPS	Functionality Usability Reliability Performance Supportability
GUI	Graphical User Interface
HLD	High Level Design
IM	Percentage of Integration Modified
IMPL	Project Implementation Phase
IFPUG	International Function Point Users Group

KSLOC	Thousands of Source Lines of Code
LCR	Life Cycle Concept Review
LEXP	Programming Language Experience
LTEX	Language and Tool Experience
LP	Learning Products
LLD	Low Level Design
MAINT	Project Maintenance Phase
MAF	Maintenance Adjustment Factor
MCF	Maintenance Change Factor
MGMT	Management
PCAP	Programming Capability
PDR	Product Design Review
PEXP	Platform Experience
PM	Project Management
PREX	Personnel Experience
PVOL	Platform Violability
QA	Quality Assurance
RELY	Required Software Reliability
REQ	Project Requirements Phase
RUSE	Required Reusability
SCED	Required Development Schedule
SEER	System Evaluation and Estimation of Resources
SEI	Software Engineering Institute
SEM	Software Estimation Model

SITE	Multi-site operation
SLIM	Software Life Cycle Model
SLOC	Source Lines of Code
SPEC	Project Specification Phase
SRR	Software Requirements Review
SU	Percentage of reuse effort due to software understanding
TIME	Execution Time Constraint
TOOL	Use of Software Tools
TM	Team Management
UFP	Unadjusted Function Points
UNFM	Programmer unfamiliarity with the software
UTC	Unit Test Criteria
WBS	Work Breakdown Structure

7. Literatura in viri

[1] Stutzke, R.D., »Software Estimation Technology: A Survey«, In Richard H. Thayer Software Engineering Project Management, IEEE Computer Society Press, pages 218-229, 1988

[2] Legg, D.B., »Synopsis of COCOMO,« In Richard H. Thayer Software Engineering Project Management, IEEE Computer Society Press, pages 230-245, 1988

[3] Gaffney, J.E., Jr., »How to estimate Software System Size,« In Richard H. Thayer Software Engineering Project Management, IEEE Computer Society Press, pages 246-256, 1988

[4] Gaffney, J.E., Jr., »How to estimate Software Project Schedules,« In Richard H. Thayer Software Engineering Project Management, IEEE Computer Society Press, pages 257-255, 1988

[5] Jones, C., »By Popular Demand: Software Estimating Rules of Thumb,« In Richard H. Thayer Software Engineering Project Management, IEEE Computer Society Press, pages 267-269, 1988

[6] Boehm, B., Clark, B., Horowitz, E., Westland C., »Cost models for future software life cycle processes: COCOMO 2.0,« In Richard H. Thayer Software Engineering Project Management, IEEE Computer Society, pages 270-307, 1988

[7] Maxwell, K.D., Applied Statistics for Software Managers, Software Quality Institute Series, Pearson Education, Publishing as Prentice Hall PTR, 2002

[8] Florac, A.W., Carleton A.C., Measuring the Software Process, The SEI Series in Software Engineering, Pearson Education, 1999

[9] Boehm, B.W., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horrowitz, E., Madachy, R., Reifer, D., Steece, B., Software Coste Estimation with COCOMO II, Prentice Hall PRT, 2002

[10] Boehm, B.W., Software Engineering Economics, Prentice Hall, 1981

[11] DeMarco, T., Controlling Software Projects, Prentice Hall PTR, 1982

[12] Pfleeger, S.L., Software Engineering Theory and Practice, Prentice Hall, 1988

[13] Verzuh, E., The Fast Forward MBA in Project Management, John Wiley and Sons, 1999

[14] Brooks, F., The Mythical Man Month, Addison Wesley, 1995

[15] Boehm, B., Abts, C., Chulani, S., »Software Development Cost Estimation Approaches«, part of Qualifying exam report at USC , 1998

[16] Solina, F., Projektno vodenje razvoja programske opreme, Založba FE in FRI, 1997

[17] »Planning Big Software Projects«, Hermes Softlab internal document, 2003

[18] »OmniBack Software Development«, Hermes Softlab internal document, 2001

[19] »Parametric Estimating Handbook«, Department of Defense, USA, 1999